Functional Programming

Final Exam

Friday, December 18 2015

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.

Stay Functional You are strictly forbidden to use mutable state (vars) or mutable collections in your solutions, unless *explicitly* required to do so.

Your Answers Answer each question on a separate sheet of paper.

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	
Total	30	

Exercise 1: Mutual Recursion in Lisp (10 points)

In this exercise, we will explore mutual recursion in the Lisp language you have seen in the labs.

Consider a new `defs` construct, which lets you provide a list of definitions instead of a single one, and which makes all definitions visible in the bodies of each other.

```
(defs ( (name1 body1) (name2 body2) (name3 body3) ... ) expr)
```

In the above, `name1`, `name2`, and `name3` are simultaneously visible in `body1`, `body2`, and `body3`. Typically bodies could be lambda definitions, which enables you to write mutually recursive functions.

Part 1: Testing even/odd numbers (3 pts)

Assume the **only** arithmetic operation you have access to is a function `minus1`, which subtracts 1 from its argument:

```
(def (minus1 x) (- x 1) ...)
```

Using the above `defs` construct, define two mutually recursive functions `odd?` and `even?` that check respectively whether a non-negative integer is odd or even:

```
(defs
  ((odd? (lambda (n) ???))
   (even? (lambda (n) ???)))
  (even? 42))
```

Note: You should not rely on a `mod` operator. You should take advantage of the fact that `odd?` can call `even?` and `even?` can call `odd?`. You are free to use basic operators of Lisp that were seen in the lab, including comparators and conditional expressions.

Hint: Remember Lisp (non-)syntax of prefix notation, for example, comparisons are written as `(= a b)`.

Note: Indent your code properly when answering this question, syntax is very important here.

Part 2: Extending the interpreter

We will now extend our Lisp interpreter so that it can handle the `defs` construct, in two steps.

Extending the environment for multiple functions (4 points)

Suppose you are given an environment `Env` and a `Map` of bindings. Implement an `updateEnvRecs` function that creates a new environment such that the corresponding `Data` is available, as needed. You can look at the `updateEnvRec` function, and the `Map` API, both in the appendix, for inspiration.

```
def updateEnvRecs(env: Env, bindings: Map[String, Data]): Env = ???
```

Extending the evaluator (3 points)

You can now extend the evaluation function `evalRec` itself. Be very careful to match the structure of `defs` correctly.

```
def evalRec(x: Data, env: Env) = x match {
  ...
  // the defs case here
  ???
}
```

Exercise 2: Streams (10 points)

As a Jedi spy accompagnied by R2D2, you arrived at the dark spaceship and you need to enter it. The spaceship is protected by a code. R2D2 can test codes but needs you to provide a stream of possible codes to brute force. Fortunately, thanks to the real force, you heard in a foreign conversation that the code is "doubled". Your first guess is that the password is an integer squared.

Square integer stream (2 points) Write a stream of squares of integers, starting from 0.

```
val squareCodes: Stream[Int] = ???
```

Exhaustive bitstring search (4 points) *Nevertheless, the door still does not open. R2D2 suggests that you write a stream of all possible binary codes.*

Write a stream of all non-empty strings of binary digits, using the constants `"0"`, `"1"` and the concatenation operation `+`. In other words, every non-empty string composed of `"0"` and `"1"` should be reached at some point. Hint: The stream can be recursive.

```
val bitCodes: Stream[String] = ???
```

Palindrome generation (2 points) *Yoda sends you a thought: "Mirrored the code is. Rising and falling, Dark forces are." It immediately comes to your mind that the code might be a binary palindrome, thus shedding light on the first conversation.*

Using the previously defined stream `bitCodes`, write a stream of all possible non-empty palindromes. Make sure to include odd-length and even-length palindromes. You may use the `.reverse` function defined on strings. You do not need to generate palindromes in increasing length order.

```
val bitPalindromeCodes: Stream[String] = ???
```

Interleaving streams (2 points) *Another Jedi sends you a stream of other possible codes which are not palindromes. The stream is finite or infinite, you do not know.*

```
val codesOfJedi: Stream[String]
```

What stream can you build for R2D2 to interleave the streams `bitPalindromeCodes` and `codesOfJedi`? Interleaving means every other element is an element of `bitPalindromeCodes`, and every other is an element of `codesOfJedi`, until there are no more elements from `codesOfJedi`. The stream then continues with codes of `bitPalindromeCodes`.

Exercise 3: The State Monad (10 points)

It is the year-end holidays. Instead of revising your exams, you want to put your newly learned Scala skills to the test, by programming a little Mario-like game.

You start off with a fairly simple design. A game is defined by running the `runGame` function on a list of `GameAction`, where you execute every action using a `doAction` function, which returns a log message describing the action you undertook:

```
def runGame(ls: List[GameAction]): List[String] = {
  "game starts" :: (ls map doAction)
}
```

There are for now 4 actions in your game:

```
sealed abstract class GameAction
case object EatMushroom extends GameAction
case object JumpOnTortoise extends GameAction
case object SkidOnBanana extends GameAction
case object FallFromBridge extends GameAction

def doAction(ga: GameAction): String = ga match {
  case EatMushroom    => "ate a mushroom"
  case JumpOnTortoise => "jumped on tortoise"
  case SkidOnBanana   => "skid on a banana"
  case FallFromBridge => "fell from bridge"
}
```

Keeping Score

You now want to add the ability to count the score accumulated through the various actions. You come up with the following scoring scheme:

- eating a mushroom awards 5 points.
- jumping on a tortoise awards 10 points.
- skidding on a banana costs 5 points.
- falling from a bridge costs 10 points.

Keeping score with mutable variables (1 point)

Suppose, **for once**, that you can use mutable variables/imperative style code, and that the sequence of actions you have is:

```
val myActions = List(EatMushroom, JumpOnTortoise, SkidOnBanana)
```

We can then implement a *side effectful* function `doAction2`, that updates the variable `score`:

```
var score = 0
val gameRun = "game starts" :: (myActions map doAction2)
//score here equals 10
```

Implement `doAction2`, such that it updates the state variable `score`:

```
def doAction2(ga: GameAction): String = ???
```

The State Monad

But we want to be purely functional in our implementation. So rather than `runGame` returning `List[String]`, it is better to have it return some sort of game state, which contains information regarding the score. To this effect, you choose to use the state monad. We represent this monad as follows:

```
final class StateM[A] private (private val makeProgress: Int => (A, Int)) {
  def runState(initState: Int): (A, Int) = makeProgress(initState)
}
```

Essentially, the state monad is a function that takes an initial state (of type `Int`, for the score) and returns a value of type `A`, and a new state. In the above signature, the `private` keywords signify, respectively, that:

- we cannot use the constructor of `StateM` outside of the class and its companion object.
- we cannot refer to `makeProgress` outside the body of `StateM` and its companion object. If we want to execute it, we will instead use the `runState` method.

To allow access and to the state, the state monad comes with two helper functions, `getState` and `putState`:

```
object StateM {
  def getState: StateM[Int] = new StateM((s: Int) => (s, s))
  def putState(newState: Int): StateM[Unit] = new StateM((s: Int) => ((), newState))
}
```

Essentially, in a purely functional world, the state monad allows us to safely store and update state variables.

Coming back to our game, we can now convert `runGame` so that it returns a `StateM[List[String]]`: the main computation of this function is the log message, but it also carries some state that represents the score:

```
def runGame3(ls: List[GameAction]): StateM[List[String]] =
  ls.foldLeft[StateM[List[String]]](unit(List("game starts"))) {
    case (prevState, newAction) =>
      for {
        msg <- prevState
        msg2 <- doAction3(newAction)
      } yield {
        msg ++ List(msg2)
      }
  }
```

We run the initial game by giving it an initial state of `0` points:

```
runGame3(ls).runState(0)
```

Desugaring For Comprehensions (2 points)

As a warmup, desugar the above for expression in terms of `flatMap`, `map` and `withFilter`:

```
for {
  msg <- prevState
  msg2 <- doAction3(newAction)
} yield {
  msg ++ List(msg2)
}
```

The monadic operations: unit (2 points)

Recall that a monad must implement the `unit` and `flatMap` operations (that respect the monad laws). Implement `unit` for `StateM`:

```
// inside the StateM companion object
def unit[A](a: A): StateM[A] = ???
```

Calling `unit(a)` returns a function that takes some initial state `s`, and returns the element `a`, along with the initial state. It simply forwards the state through.

The monadic operations: flatMap (3 points)

Implement `flatMap`, or the monadic bind, for `StateM`:

```
// inside StateM[A], the class
def flatMap[B](f: A => StateM[B]): StateM[B] = ???
```

Calling `sm flatMap f` returns an instance of the state monad where, upon passing an initial state, the computation of `sm` is executed yielding a pair `(a, s2)`, of type `(A, Int)`. The value `a` is then passed to `f`, yielding an instance of `StateM[B]`. We finally pass `s2` to this instance.

Hint 1: here is the implementation of `map` for `StateM`:

```
// in StateM[A], the class
def map[B](f: A => B): StateM[B] = {
  val res = { (s: Int) =>
    val (a, s2) = makeProgress(s)
    (f(a), s2)
  }
  new StateM(res)
}
```

Hint 2: follow the types!

Actions that keep scores (2 points)

Finally, we need to implement `doAction3`. Convert the `doAction` function so that it now returns a `StateM[String]` that respects the above scoring function. Remember, you are only allowed to use the monad operations (`unit`, `map`, `flatMap`), `getState` and `putState`:

```
def doAction3(ga: GameAction): StateM[String] = ???
```

Appendix

A non-exhaustive (but useful) API

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

on `List` (containing elements of type `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.toMap: Map[K, V]`: provided `A` is a pair `(K, V)`, converts this list to a `Map[K, V]`.
- `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)]`.

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

on `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`.

on `Stream` (the object):

- `Stream.from(i: Int): Stream[Int]`: Creates an infinite stream of integers starting at i.

on `Map` (containing keys of type `K`, values of type `V`):

- `mp.get(k: K): Option[V]`: For a given key `k`, returns `Some(v)` if a value exists in the map `mp`, `None` otherwise.

Lisp interpreter:

```
/** The environment, as a function */
type Env = String => Option[Data]

/** functions that update the environment */
def updateEnv(env: Env, bindings: List[(String,Data)]): Env = bindings match {
  case Nil => env
  case (id, d) :: rest => ((x: String) =>
      if (x == id) Some(d)
      else updateEnv(env, rest)(x))
}

def updateEnvRec(env: Env, s: String, expr: Data): Env = {
  def newEnv: Env = ((id: String) =>
    if (id == s) Some(evalRec(expr, newEnv))
    else env(id)
  )
  newEnv
}

/** The evaluation function */
def evalRec(x: Data, env: Env): Data = x match {
  case i: Int => i

  case Symbol(s) => env(s) match {
    case Some(v) => v
    case None => sys.error("Unknown symbol " + s)
  }
  case List('lambda, params: List[Data], body) =>
    ((args: List[Data]) => {
      val paramBinding = params.map(_.asInstanceOf[Symbol].name).zip(args)
      evalRec(body, updateEnv(env, paramBinding))
    })

  case List('val, Symbol(s), expr, rest) =>
    evalRec(rest, updateEnv(env, List(s -> evalRec(expr, env))))

  case List('def, Symbol(s), expr, rest) => {
    evalRec(rest, updateEnvRec(env, s, expr))
  }

  case List('if, bE, trueCase, falseCase) =>
    if (evalRec(bE, env) != 0) evalRec(trueCase, env)
    else evalRec(falseCase, env)

  case opE :: argsE => {
    val op = evalRec(opE, env).asInstanceOf[List[Data] => Data]
    val args: List[Data] = argsE.map((arg: Data) => evalRec(arg, env))
    op(args)
  }
}
```