Functional Programming

Final Exam Solution

Friday, December 18 2015

Exercise 1: Mutual Recursion in Lisp (10 points)

Part 1

```
(def*
  ((even? (lambda (x) (if (= x 0) 1 (odd? (minus1 x)))))
   (odd? (lambda (x) (if (= x 0) 0 (even? (minus1 x))))))
  (even? 42))
```

Part 2

Extending the environment for multiple functions

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

  def newEnv: Env = (id: String) =>
    bindings.get(id) match {
      case None => env(id)
      case Some(expr) => Some(evalRec(expr, newEnv))
    }

  newEnv

}
```

Extending the evaluator

```
def evalRec(x: Data, env: Env): Data = x match {
  ...
  //the defs case
  case List('defs, functions, rest) =>
    val bindings = functions.asInstanceOf[List[Data]]
    evalRec(
      rest,
      updateEnvRecs(env, (bindings map { case List(Symbol(s), expr) => (s, expr)}).toMap)
    )
}
```

Exercise 2: Streams (10 points)

Square integer stream: (2 points) Possible solutions:

```
val squareCodes: Stream[Int] = Stream.from(0).map(x => x*x)
val squareCodes: Stream[Int] = for(x <- Stream from 0) yield x*x
val squareCodes: Stream[Int] = {
  def p(i: Int): Stream[Int] = (i*i) #:: p(i+1)
  p(0)
}
def squareCodes(i: Int = 0): Stream[Int] = (i*i) #:: squareCodes(i + 1)
```

Exhaustive bitstring search: (2 points) Concatenation may be reversed; inner sequences constructors can be `Stream`, `List` or anything.

```
val bitCodes: Stream[String] = "0" #:: "1" #:: bitCodes.flatMap(s => Stream("0" + s, "1" + s))
val bitCodes: Stream[String] = "0" #:: "1" #:: (for(s <- bitCodes) yield Stream("0" + s, "1" + s)
val bitCodes: Stream[String] = "0" #:: "1" #:: (for(s <- bitCodes
                                                c <- List("0", "1")) yield (c + s))
val bitCodes: Stream[String] = "0" #:: "1" #:: (for{s <- bitCodes
                                                c <- List("0", "1")
                                                b = c + s} yield b)
```

Wrong solutions:

```
// Wrong: It will yield 0, 1, 00, 01, 000, 001, 0000, 0001 ....
val bitCodes: Stream[String] = "0" #:: "1" #:: (for(c <- List("0", "1")
                                                s <- bitCodes) yield (c + s))
```

Palindrome generation: (2 points) Possible solutions:

```
val bitPalindromeCodes: Stream[String] = bitCodes.flatMap(s =>
    Stream(s + s.reverse, s + "0" + s.reverse, s + "1" + s.reverse))
val bitPalindromeCodes: Stream[String] = (for(s <- bitCodes) yield
    Stream(s + s.reverse, s + "0" + s.reverse, s + "1" + s.reverse)).flatten
val bitPalindromeCodes: Stream[String] = (for(s <- bitCodes
    c <- List(s + s.reverse, s + "0" + s.reverse, s + "1" + s.reverse)) yield c
```

Bonus: Interleaving streams: Possible solution:

```
def interleavedCodes(s: Stream[String], t: Stream[String]): Stream[String] =
  if(t.isEmpty) s else s.head #:: t.head #:: interleavedCodes(s.tail, t.tail)
interleavedCodes(bitPalindromeCodes, codesOfJedi)
```

Wrong solution:

```
// Will crash R2D2 if the stream t is not infinite.
def interleavedCodes(s: Stream[String], t: Stream[String]): Stream[String] =
  s.head #:: t.head #:: interleavedCodes(s.tail, t.tail)
interleavedCodes(bitPalindromeCodes, codesOfJedi)
```

Exercise 3: The State Monad (10 points)

This question is inspired from an example on the HaskellWiki. Check it out here:

```
https://wiki.haskell.org/State_Monad#Complete_and_Concrete_Example_1
```

Keeping score with mutable variables (1 point)

We first create a helper function, `scoreFunction`:

```
def scoreFunction(ga: GameAction, score: Int): Int = ga match {
  case EatMushroom    => score + 5
  case JumpOnTortoise => score + 10
  case SkidOnBanana   => score - 5
  case FallFromBridge => score - 10
}
```

We can then write the following imperative code:

```
def doAction2(ga: GameAction): String = {
  score = scoreFunction(ga, score)
  doAction(ga)
}
```

Desugaring For Comprehensions (2 points)

Doing the desugaring mechanically, we get:

```
acc flatMap { msg =>
  doAction2(elem) map { msg2 => msg ++ List(msg2) }
}
```

The monadic operations: unit (2 points)

```
def unit[A](a: A) = new StateM((s: Int) => (a, s))
```

The monadic operations: flatMap (3 points)

```
def flatMap[B](f: A => StateM[B]): StateM[B] = {
  val res = (s: Int) => {
    val (a, s2) = makeProgress(s)
    f(a).makeProgress(s2)
  }

  new StateM(res)
}
```

Actions that keep scores (2 points)

```
def doAction3(ga: GameAction): StateM[String] = for {
  score <- getState
  _ <- putState(scoreFunction(ga, score))
} yield doAction(ga)
```