Programming Principles

Midterm Solution

Friday, November 9th 2012

Exercise 1: For-Comprehensions and De-Sugaring (10 points)

```
type Result = (String, Int)
type Quiz = List[Result]
type Course = List[Quiz]
```

Duels and Higher-Order Functions (2 points)

```
def duels(dwarves: List[String]): List[Duel] = dwarves.flatMap { dwarf1 =>
  dwarves.filter(_ != dwarf1).map { dwarf2 =>
    (dwarf1, dwarf2)
  }
}
```

Comprehending the Function

```
def duels(dwarves: List[String]): List[Duel] =
  for (
    dwarf1 <- dwarves
    dwarf2 <- dwarves
    if dwarf1 != dwarf2
  ) yield (dwarf1, dwarf2)
```

Recursion Galore

```
def duels3(dwarves: List[String]): List[Duel] = dwarves match {
  case x :: y :: Nil => (x,y) :: (y,x) :: Nil
  case x :: y :: xs => (duels3(x :: y :: Nil) ++ duels3(x :: xs) ++ duels3(y :: xs)).distinct
}
```

Picking the Winner

```
def winner(duels: List[(Duel, Int)]): String = {
  val victoriousDwarves = duels.map{ case ((dwarf1, dwarf2), i) =>
    if(i == 1) dwarf1 else dwarf2
  }

  val dwarvesToVictories = victoriousDwarves.groupBy(x => x).map{ case (dwarf, victories) =>
    (dwarf, victories.size)
```

```
  }

  val sorted = dwarvesToVictories.toList.sortWith{
    case (d1, d2) => d1._2 > d2._2
  }

  sorted.head._1
}
```

Exercise 2: Lisp (10 points)

Implementing `filter` in Lisp

```
def (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst)) (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

Desugaring `and` in Lisp

def (desugar-and lst) (cons (quote if) (cons (car (cdr lst)) (cons (car (cdr (cdr lst))) (cons 0 nil))))

Exercise 3: Memoization (10 points)

The `fib` method

```
def fib(n: Int): Int = if (n <= 2) 1 else fib(n - 1) + fib(n - 2)
```

The `memo` operation

```
def memo[A, B](f: A => B): A => B = {
  val cache = mutable.Map[A, B]()
  (a: A) => {
    cache.get(a) match {
      case Some(b) =>
        b
      case None =>
        val b = f(a)
        cache.put(a, b)
        b
    }
  }
}
```

Improving the complexity using the `memo` operation

```
val f: Int => Int = memo { n: Int =>
  if (n <= 2) 1 else f(n - 1) + f(n - 2)
}

def memofib(n: Int) = f(n)
```