Scala Language 02 Hands On

# For Comprehensions

**Problem 1**

Write a for comprehension to take the first element of a list of lists. E.g. given

val l = List(List("one","two","three"), List("Apple","Orange","Pear"), List("Apple","Intel","Microsoft"))

write a for comprehension to produce

List(one, Apple, Apple)

Now write a for comprehension to produce lists starting with “Apple”

**Problem 2**

Enter

val l1 = List(“one”, “two”, “three”)

val l2 = List(“a”, “b”, “c”)

Write a for comprehension to produce the cross product of l1 and l2. The result should be

List((a,one), (a,two), (a,three), (b,one), (b,two), (b,three), (c,one), (c,two), (c,three))

# Higher Order Functions

**Problem 3**

Use map to take the first element of a lsit of lists. E.g., given

val l = List(List("one","two","three"), List("Apple","Orange","Pear"), List("Apple","Intel","Microsoft"))

use map to produce

List(one, Apple, Apple)

Now use filter to produce lists starting with “Apple”

**Problem 4**

The tree

1

/ \

2 3

/ \

4 5

Can be represented in Scala with the following code

case class Tree(x: Int, children: List[Tree])

val tree = Tree(1,List(Tree(2,Nil),Tree(3,List(Tree(4,Nil),Tree(5,Nil)))))

Enter the above code and execute

var nodes = List(tree)

what happens when you execute

nodes = nodes.flatMap(\_.children)

Do it twice more. Can you describe the effect in abstract terms?

**Problem 5**

Enter

val l1 = List(“one”, “two”, “three”)

val l2 = List(“a”, “b”, “c”)

Execute

l1 zip l2

What happened?

**Problem 6**

Enter

val l1 = List(“one”, “two”, “three”)

val l2 = List(“a”, “b”, “c”)

Write an expression using flatMap and map to produce the cross product of l1 and l2. The result should be

List((a,one), (a,two), (a,three), (b,one), (b,two), (b,three), (c,one), (c,two), (c,three))

**Problem 7**

Execute the following

val cars = List("Mazda", "BMW", "Mercedes", "Bugati", "Honda", "Toyota")

cars groupBy { \_.charAt(0) }

What happened? See [the scaladoc](http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.Map) for documentation for groupBy.

# 

# Trait Linearization

The ability of types to inherit multiple traits introduces multiple inheritance into scala. Scala uses an algorithm to determine the order in which super() gets called in a type DAG. The algorithm is as follows:

* A pre-order depth-first-search is done on the hierarchy, bottom-to-top, right-to-left, generating a list of types
* The type list is scanned left-to-right, and a type is removed from the list if it appears again to the right

**Problem 8**

Consider the following code

class Animal {

def printMe() { println("I'm an Animal!") }

}

trait Furry extends Animal {

override def printMe() { println("I'm Furry!"); super.printMe() }

}

trait HasLegs extends Animal {

override def printMe() { println("I have legs!"); super.printMe() }

}

trait FourLegged extends HasLegs {

override def printMe() { println("I'm four-legged!"); super.printMe() }

}

// Multiple inheritance with the keyword "with"

class Cat extends Animal with Furry with FourLegged {

override def printMe() { println("I'm a cat!"); super.printMe() }

}

Given the linearization algorithm, what do you predict the output will be when you call

new Cat().printMe()?

Execute the code and see. See the Trait Linearization Slide in [the Slide deck](http://goo.gl/HMhdBd) for a visual of what’s going on.

# Pattern Matching

**Problem 9**

Enter the following class hierarchy

abstract class Person(val name: String)

class Employee(override val name: String, val salary:Int) extends Person(name)

case class Engineer(override val name: String, override val salary:Int, languages:List[String]) extends Employee(name, salary)

case class Recruiter(override val name: String, override val salary:Int, recruits:List[Engineer]) extends Employee(name, salary)

and the pattern matching function

def peopleMatcher(p: Person) = p match {

case Recruiter(name, salary, people) => f"Recruiter $name has ${people.size} people"

case Engineer(name, salary, Nil) => f"$name has no skills!"

case Engineer(name, salary, languages) if languages.contains("Scala") =>

val x = f"$name Knows Scala!"

x

case Engineer(name, salary, languages) => f"$name Doesn't know Scala :("

case \_ => "Huh?"

}

Create people of various Ilks (recruiters, employees, and regular employees) and match them using peopleMatcher.