BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
First Semester 2018-19
Principles of Programming Languages
Lab Session 2: Functional Programming in Scala

Higher Order Functions

When writing purely functional programs, we'll often find it useful to write a function that accepts other functions as arguments. This is called a higher-order function (HOF).

Motivation for Higher order Functions

- Take the sum of the integers between a and b

```
def sumInts(a: Int, b: Int): Int =
        if (a > b) 0 else a + sumInts(a + 1, b)
```

- Take the sum of the cubes of all the integers between a and b

```
def cube(x: Int): Int = x * x * x

def sumCubes(a: Int, b: Int): Int =
        if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

- Take the sum of the factorials of all the integers between a and b

```
def sumFactorials(a: Int, b: Int): Int =
  if (a > b) 0 else factorial(a) + sumFactorials(a + 1, b)
```

Note: You already know how to define the factorial() function.

Note how similar these methods are. Can we factor out the common pattern?

Lets define sum

```
def sum(f: Int => Int, a: Int, b: Int): Int =
if (a > b) 0
else f(a) + sum(f, a + 1, b)
```

Note: Look at the syntax for the function 'sum', one its input is a function 'f' which takes as input an int and returns an int.

We can then use

```
def id(x: Int): Int = x
def sumInts(a: Int, b: Int) = sum(id, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(factorial, a, b)
```

So far we've defined only monomorphic functions, or functions that operate on only one type of data. But in the current world you cannot assume anyone's return type. We want to write code that works for any type it's given. These are called polymorphic functions.

The following function returns the first index in an array where the key occurs, or -1 if it's not found. This function is specific to 'string' datatype.

New :

```
def findFirst(ss: List[String], key: String,index: Int): Int = ss match {
        Case Nil : -1
        Case x::xs => x match {
                        Case key : index
                        Case _ : findFirst(xs, key, index+1)
                }

}
```

To celebrate diversity of return types, we define a polymorphic function

New :

```
def findFirst[A] (ss:Array[A], p: A => Boolean ,index: Int): Int = ss match {
        Case Nil : -1
        Case x::xs => p(x) match {
                        True => index
                        False => findFirst(xs, key, index+1)
                }

}
```

Here rather than assuming the datatype as 'String', we abstract over the datatype. Here you will have to define the function p. (Why? Equality is a subjective topic, while equating strings maybe you don't want to consider numbers, exclamation marks. Therefore 'Arsenal' should equate to 'Arsenal?' , 'Arse!nal' , Arsenal4'.)

Anonymous functions

Writing a seperate definition for every function can be tiring after some time. Maybe you just wanted to use the square or cube function in just this one place. Anonymous functions can help you.

How will you use the polymorphic “findFirst” function that you created above??

def equalInt9…… → Avoid defining functions that are used only once. Try anonymous functions instead.

findFirst(Array(20, 0, 2), (x: Int) => x == 2) → For int

findFirst(Array(‘377’,’remove’,’Section’), (x: String) => x== ‘okay’) → For String

Note : The syntax (x: Int) => x == 9 is a function literal or anonymous function. Instead of defining this function as a method with a name, we can define it inline using this convenient syntax. This particular function takes one argument called x of type Int , and it returns a Boolean indicating whether x is equal to 9 .

Using anonymous functions for finding square and cubes

```
def sumInts(a: Int, b: Int) = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
```

Exercises

1) Create a function which constructs a new list by adding two lists by adding their corresponding elements. For example, List(3,0,1) and List(4,3,2) become List(7,3,3). Generalize the function so that it is not specific to integer or addition.

2) Define a Scala procedure “filter-list”, which takes a predicate and a list as arguments, and returns a list that contains the elements of the given list that satisfy the given predicate (or condition). Use this to remove all even numbers from a list

3) Concatenate a list of lists into single list

4) Check if a given list is sorted

5) Implement Recursive quicksort