19CSE313 – PRINCIPLES OF PROGRAMMING LANGUAGES

Functions, Closures, Higher order Functions

FIRST-CLASS FUNCTIONS

- Defining functions as unnamed *literals* and then passing them around as *values*
- A function literal is compiled into a class that when instantiated at runtime is a *function value*
- Function literals exist in the source code,whereas function values exist as objects at runtime
- The distinction is much like that between classes (source code) and objects (runtime)
- Example:

```
scala> (x: Int) => x + 1

val res0: Int => Int = $Lambda$1006/891327883@6cc64028
```

=> designates that this function converts the thing on the left (any integer x) to the thing on the right (x + 1)

FUNCTION VALUES

- Function values are objects and can be stored in variables
- Since they are functions, they can be invoked using the usual parentheses function-call notation

- scala> var increase = (x: Int) => x + 1
- var increase: Int => Int = $Lambda$1010/1600427200@6826b70f

- scala> increase(10)
- val res1: Int = 11

ASSIGNING DIFFERENT VALUES

- Because increase, in this example, is a var, a different function value can be assigned later on.
- Example:

- scala> increase = (x: Int) => x + 9999
- // mutated increase

- scala> increase(10)
- val res2: Int = 10009

MORE THAN ONE STATEMENT IN THE FUNCTION LITERAL

```
scala> increase = (x: Int) => {
     | println("We")
     | println("are")
     | println("here!")
     | x + 1
     | }
// mutated increase


scala> increase(10)
We
are
here!
val res3: Int = 11
```

- To have more than one statement in the function literal, surround its body by curly braces and put one statement per line, thus forming a block.
- Just like a method, when the function value is invoked, all of the statements will be executed, and the value returned from the function is whatever results from evaluating the last expression.

EXAMPLES:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
val someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
10
```

foreach takes a function as an argument and invokes that function on each of its elements

FILTER METHOD FOR COLLECTIONS

```
scala> someNumbers
val res5: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.filter((x: Int) => x > 0)
val res6: List[Int] = List(5, 10)
```

SHORT FORMS OF FUNCTION LITERALS – TARGET TYPING

- leave off the parameter types

```
scala> someNumbers.filter((x) => x > 0)
val res7: List[Int] = List(5, 10)
```

- leave out parentheses around a parameter whose type is inferred

```
scala> someNumbers.filter(x => x > 0) //
val res8: List[Int] = List(5, 10)
```

PLACEHOLDER SYNTAX

- To make a function literal even more concise, you can use underscores as placeholders for one or more parameters, so long as each parameter appears only one time within the function literal.

```
scala> someNumbers.filter(_ > 0)          //equivalent to x => x > 0
val res9: List[Int] = List(5, 10)
```

UNDERSCORES AS PLACEHOLDERS FOR PARAMETERS

```
scala> val f = _ + _
           ^
       error: missing parameter type for expanded function ((<x$1: error>, x$2) =>
x$1.$plus(x$2))
                ^
       error: missing parameter type for expanded function ((<x$1: error>, <x$2: error>) =>
x$1.$plus(x$2))
```

```
scala> val f = (_: Int) + (_: Int)
val f: (Int, Int) => Int = $Lambda$1071/6506
```

```
scala> f(5, 10)
val res10: Int = 15
```

Note: _ + _ expands into a literal for a function that takes two parameters. Use this short form only if each parameter appears in the function literal exactly once. Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly. The first underscore represents the first parameter, the second underscore the second parameter, the third underscore the third parameter, and so on.

PARTIALLY APPLIED FUNCTIONS

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
def sum(a: Int, b: Int, c: Int): Int

scala> sum(1,2,3)
val res11: Int = 6

scala> val a = sum _          // a refers to a function value object
val a: (Int, Int, Int) => Int = $Lambda$1082/862366104@3c0403a

scala> a(1,2,3)               //apply method takes three arguments
val res13: Int = 6
```

```
scala> a.apply(1, 2, 3)
res12: Int = 6
```

SUPPLYING ONLY *SOME* OF THE REQUIRED ARGUMENTS

```
scala> val b = sum(1, _: Int, 3)
val b: Int => Int = $Lambda$1093/945470472@78067cc4

scala> b(2)
val res15: Int = 6

scala> b(5)
val res16: Int = 9
```

PARTIAL APPLICATION

```
object Partial1
{
 def main(args:Array[String])
 {
   val sum = (a: Int, b: Int, c: Int) => a + b + c
   val f = sum(10,_:Int,30)
   println(f(20));
 }
}
```

CLOSURES

```
scala> (x: Int) => x + more
                     ^
       error: not found: value more
```

- This function adds "more" to its argument, but what is more?
- From the point of view of this function, more is a *free variable* because the function literal does not itself give a meaning to it.
- The x variable, by contrast, is a *bound variable* because it does have a meaning in the context of the function: it is defined as the function's lone parameter, an Int.

CLOSURES

```
scala> var more = 1

var more: Int = 1


scala> val addMore = (x: Int) => x + more

val addMore: Int => Int = $Lambda$1102/1795982619@2351ad5a


scala> addMore(10)

val res18: Int = 11
```

The function value (the object) that's created at runtime from this function literal is called a *closure*.

MODIFYING THE CLOSURE

```
scala> more = 9999

// mutated more


scala> addMore(10)

val res19: Int = 10009
```

Scala's closures capture variables themselves, not the value to which variables refer

HIGHER ORDER FUNCTION – EXAMPLE 1

```
object higher1
{
def math(x: Double, y:Double, fn: (Double,Double)=>Double): Double = fn(x,y);
 def main(args:Array[String])
 {
   val result = math(50,20,(x,y)=>x max y);
   println(result);
 }
 }
```

HIGHER ORDER FUNCTION – EXAMPLE 2

```
object higher2
{
def math(x: Double, y:Double, z:Double, fn: (Double,Double)=>Double):
Double = fn(fn(x,y),z);
 def main(args:Array[String])
 {
   val result = math(50,20,10,(x,y)=>x * y);
   println(result);
 }
}
```

CURRYING IN SCALA

- Technique of transforming a function that takes multiple arguments to a function that takes a single argument.

```
object curry
{
 def add(x:Int,y:Int) = x + y
 def add2(x:Int) = (y:Int) => x + y; //curried version
 def add3(x:Int)(y:Int) = x + y;   //Simpler scala version
 def main(args:Array[String])
 {
  println(add(20,10));
  println(add2(20)(10));          //calling  curried add2
  val sum40 = add2(40);
  println(sum40(100));           //Partial application
  println(add3(100)(200));       //calling simpler scala version
  val sum50 = add3(50)_;         //partial application of simple version
  println(sum50(400));
}
}
```

```
D:\PPL\Scala>
scala curry
30
30
140
300
450
```

MAP

```
object mymap
{

 val lst = List(1,2,3,5,7,10,13);

 def main(args:Array[String])
 {
   println(lst.map(_ * 2));                    //method 1
   println(lst.map(x => x + 3));               //method 2 - using nameless function

   println(lst.map(x => "hi"+ x));             //string concatenation
   println(lst.map(x => "hi" * x));            //string multiplication

   println("hello".map(_.toUpper));      //String is a list of characters
 }
}
```

```
D:\PPL\Scala>scala mymap
List(2, 4, 6, 10, 14, 20, 26)
List(4, 5, 6, 8, 10, 13, 16)
List(hi1, hi2, hi3, hi5, hi7, hi10, hi13)
List(hi, hihi, hihihi, hihihihihi, hihihihihihihi,
hihihihihihihihihihi, hihihihihihihihihihihihi)
HELLO
```

FLATTENING A LIST OF LISTS

```
object flatten
{
  def main(args:Array[String])
 {
  println(List(List(1,2,3),List(4,5,6)).flatten);
 }
}
```

```
D:\PPL\Scala>scala flatten
List(1, 2, 3, 4, 5, 6)
```

FLATMAP

Maps and Flattens the list.

```
object flatmap
{

 val lst = List(1,2,3,5,7,10,13);

 def main(args:Array[String])
 {
  println(lst.flatMap(x=>List(x,x+1)));
  println(lst.map(x=>List(x,x+1)));
 }

}
```

```
D:\PPL\Scala>scala flatmap
List(1, 2, 2, 3, 3, 4, 5, 6, 7, 8, 10, 11, 13, 14)
List(List(1, 2), List(2, 3), List(3, 4), List(5, 6), List(7, 8), List(10, 11), List(13, 14))
```

FILTER

```
object myfilter
{

 val lst = List(1,2,3,5,7,10,13);

 def main(args:Array[String])
 {
  println(lst.filter(x=>x%2==0));
  println(lst.filter(x=>x%2!=0));
 }

}
```

```
D:\PPL\Scala>scala myfilter
List(2, 10)
List(1, 3, 5, 7, 13)
```

REDUCE LEFT / RIGHT EXAMPLE

- Takes an associative binary operator function as a parameter

```
object myreduce
{

 val lst = List(1,2,3,5,7,10,13);
 val lst2 = List("A","B","C");
 def main(args:Array[String])
 {
    println(lst.reduceLeft(_+_));  //adds all the elements of lst
    println(lst.reduceLeft((x,y)=>{println(x + "," + y);x+y;}))
    println(lst2.reduceLeft(_+_));  //concatenates lst2
    println(lst.reduceRight(_+_));  //adds all the elements of lst
    println(lst.reduceLeft(_-_));  //subtract using reduceRight
    println(lst.reduceRight(_-_)); //subtract using reduceRight
    println(lst.reduceLeft((x,y)=>{println(x + "," + y);x-y;}))
    println(lst.reduceRight((x,y)=>{println(x + "," + y);x-y;}))
 }
}
```

```
D:\PPL\Scala>scala myreduce
41
1,2
3,3
6,5
11,7
18,10
28,13
41
ABC
41
-39
7
1,2
-1,3
-4,5
-9,7
-16,10
-26,13
-39
10,13
7,-3
5,10
3,-5
2,8
1,-6
7
```

FOLD / SCAN - LEFT / RIGHT - EXAMPLE

- Fold: Similar to reduceLeft/reduceRight but initial arguments cen be passed in folds.
- Scan – Prints the intermediate results.

```
object myfoldscan
{
 val lst = List(1,2,3,5,7,10,13);
 val lst2 = List("A","B","C");

def main(args:Array[String])
 {
    println(lst.foldLeft(0)(_+_));  //adds all the elements of lst
    println(lst.foldLeft(100)(_+_));  //adds all the elements of lst

    println(lst2.foldLeft("z")(_+_));  //concatenates

    println(lst.scanLeft(100)(_+_));  //adds all the elements of lst
    println(lst2.scanLeft("z")(_+_));  //concatenates
 }
}
```

```
D:\PPL\Scala>scala
myfoldscan
41
141
zABC
List(100, 101, 103, 106, 111,
118, 128, 141)
List(z, zA, zAB, zABC)
```

THANK YOU