PROGRAMMING IN SCALA

Working with lists

Displaying lists

```
val myList = List(1,2,3,4,5)

//toString returns canonical string representation
myList.toString

// mkString has 4 operands:
 the list to be displayed,
 a prefix string ‘pre’
 a separator string ‘sep’
 a postfix string ‘post’

myList mkString (“[”, “,”, “]”)
[1,2,3,4,5]
```

Displaying lists

```
val myList = List(1,2,3,4,5)

// mkString variant taking only separator string
myList mkString sep equals myList mkString (“”, sep, “”)

myList mkString “” → “12345”

// mkString variant taking no arguments
myList mkString equals myList mkString “”

myList mkString → “12345”
```

Higher-order methods on class List

```
val myList = List(1,2,3,4,5)
val yourList = List(“the”, “quick”, “brown”, “fox”)

// mapping over lists
myList map (_ + 1)   → List(2,3,4,5,6)
yourList map (_.length) → List(3,5,5,3)

// filtering lists
myList filter (_ %2 == 0) → List(2,4)
yourList filter (_.length == 3) → List(“the”, “fox”)
```

Higher-order methods on class List

```
val myList = List(1,2,3,4,5)
val yourList = List(“the”, “quick”, “brown”, “fox”)

//predicates over lists

myList forall (_ > 0)  → True
yourList exists (_.length == 3) →True


def hasZeroRow(m: List[List[Int]]) =
  m exists (row => row forall (_ == 0))

val diag3 = (List(List(1,0,0), List(0,1,0), List(0,0,1))

hasZeroRow(diag3) → False
```

Mapping over lists

- `flatMap` is similar to `map` but takes a function returning a list of elements as right-hand operand.
- Applies the function to each list element and returns the concatenation of all function results.

```
val numbers = List(1,2,3,4,5)
val words = List(“the”, “quick”, “brown”, “fox”)

words map (_.toList)  →
List(List(t,h,e), List(q,u,i,c,k), List(b,r,o,w,n), List(f,o,x))

words flatMap (_.toList)  →
List(t,h,e,q,u,i,c,k,b,r,o,w,n,f,o,x)

Words.map(_.toUpperCase)
Seq[String] = List(THE, QUICK, BROWN, FOX)

fruits.flatMap(_.toUpperCase)
Seq[Char] = List(T,H,E,Q,U,I,C,K,B,R,O,W,N,F,O,X)
```

Mapping over lists

- You can also create a list using Range.
- Similar to range function in Python ..
  - Can be of a range of value between x ➔ y
  - Can also iterate with a defined number of steps (by how many?)

```
List.range(1,5)

List[Int] = List(1, 2, 3, 4)


List.range(1,10,2)

List[Int] = List(1, 3, 5, 7, 9)
```

Mapping over lists

flatmap is similar to map but takes a function returning a list of elements as righthand operand. Applies the function to each list element and returns the concatenation of all function results.

```
List.range(1,5) map
          (i => List.range(1,i) map (j => (i,j))

i = 1 → ()
i = 2 → ((2,1))
i = 3 → ((3,1),(3,2))
i = 4 → ((4,1), (4,2), (4,3))

map→ List(List(), List((2,1)), List((3,1), (3,2)),
List((4,1), (4,2), (4,3)))
```

Mapping over lists

flatmap is similar to map but takes a function returning a list of elements as righthand operand. Applies the function to each list element and returns the concatenation of all function results.

```
List.range(1,5) flatMap
          (i => List.range(1,i) map (j => (i,j))

i = 1 → ()
i = 2 → ((2,1))
i = 3 → ((3,1),(3,2))
i = 4 → ((4,1), (4,2), (4,3))

flatmapping → List((2,1),(3,1),(3,2),(4,1),(4,2),
(4,3))
```

Filtering lists

filter takes a list and a predicate as operands,
returns the list of all elements for which the predicate holds.

```
(xs: List[T]) filter (p:T => Boolean)

val words = List("the", "quick", "brown", "fox")
words filter (_.length == 3) → List(the, fox)


List(1,2,3,4) filter (_ >2) →  List(3, 4)
```

filterNot is the opposite of Filter

```
(xs: List[T]) filterNot (p:T => Boolean)

val words = List("the", "quick", "brown", "fox")

words filter (_.length == 3) → List(quick)
List(1,2,3,4) filterNot (_ >2) →  List(1, 2)
```

Filtering lists

partition returns a pair of lists:
one list contains the elements for which predicate is **True**,
the other contains all elements for which predicate is **False.**

```
(xs: List[T]) partition (p:T => Boolean)

val words = List(“the”, “quick”, “brown”, “fox”)

words partition (_.length == 3)  →
      (List(the, fox), List(quick, brown))

List(1,5,10,20) partition (_ %2==0) → (List(10,
20),List(1, 5))
```

Filtering lists

`filter` takes a list and a predicate as operands,
returns the list of all elements for which the predicate holds.
`partition` returns a pair of lists:
one list contains the elements for which predicate is True,
the other contains all elements for which predicate is False.

```
 (xs: List[T]) filter (p:T => Boolean)
 (xs: List[T]) partition (p:T => Boolean)

xs partition p(_) equals (xs filter p(_), xs filter !p(_))
```

Filtering lists

takeWhile is similar to take but takes predicate as right hand operand
returns longest prefix for which predicate holds ➔ takes with condition.
dropWhile similar to drop
returns all except longest prefix for which predicate holds

```
(xs: List[T]) takeWhile (p:T => Boolean)
(xs: List[T]) dropWhile (p:T => Boolean)


val words = List(“the”, “quick”, “brown”, “fox”)

words takeWhile (_.length == 3) → List(the)

words dropWhile (_.length == 3)  →
                                  List(quick, brown, fox)
```

Filtering lists

takeRight Returns the rightmost n elements from this list.
dropRight Returns the list wihout its rightmost n elements.

```
(xs: List[T]) takeWhile (p:T => Boolean)
(xs: List[T]) dropWhile (p:T => Boolean)


val words = List(“the”, “quick”, “brown”, “fox”)

List(10,20,30,50,55,66) takeRight (3) → List(50, 55, 
66)
 List(10,20,30,50,55,66) dropRight (3) →  List(10, 
20, 30)
```

Filtering lists

```
xs span p equals (xs takeWhile p, xs dropWhile p)

val numbers = List(1, 2, 3, -4, -5)

numbers span (_ > 0) → (List(1,2,3), List(-4,-5))
```

Folding

- Folding is a very powerful operation on Scala Collections
- Scala.collection has three folding mechanisms:

```
    Fold:
    foldLeft
    foldRight
```

```
def fold[A1 >: A](z: A1)(op: (A1, A1) ⇒ A1): A1

def foldLeft[B](z: B)(op: (B, A) ⇒ B): B

def foldRight[B](z: B)(op: (A, B) ⇒ B): B
```

- process a data structure recursively through use of a pre-defined combining operation **op** and an initial value **z** , then gives a return value.

Folding

/: “foldLeft”

```
(z /: xs) (op)
/* three objects:
     start value (z)
     a list (xs)
     binary operation (op)
*/

(z /: List(a,b,c)) (op) equals op(op(op(z,a), b), c)

    in infix form equals (((z (op) a) (op) b) (op) c)
```

Folding

/:   “foldleft”

```
def sum(xs:List[Int]): Int = (0 /: xs) (_ + _)

sum(List(2,3,6)) → (((0 + 2) + 3) + 6)

def product(xs:List[Int]): Int = (1 /: xs) (_ * _)

product(List(2,3,6)) → (((1 * 2) * 3) * 6)
```

Question: how to get the size of all String in a list using folding?

Folding

/: “foldleft”

```
def sum(xs:List[Int]): Int = (0 /: xs) (_ + _)

sum(List(2,3,6)) → (((0 + 2) + 3) + 6)

def product(xs:List[Int]): Int = (1 /: xs) (_ * _)

product(List(2,3,6)) → (((1 * 2) * 3) * 6)
```

Question: how to get the size of all String in a list using folding?

Folding

:\ “foldright”

```
(xs :\ z) (op)
/* three objects:
      start value (z)
      a list (xs)
      binary operation (op)
*/

(List(a,b,c) :\ z) (op) equals op(a, op(b, op(c,z)))

    in infix form equals (a (op) (b (op) (c (op)
z)))
```

Folding lists

```
def flattenLeft[T](xss: List[List[T]]) =
    (List[T]() /: xss) (_:::_)

flattenLeft(List(List(1,2), List(3,4), List(5,6)))
→ (List(1,2) ::: List(3,4)) ::: List(5,6)


def flattenRight[T](xss: List[List[T]]) =
    (xss :\ List[T]()) (_:::_)

flattenRight(List(List(1,2), List(3,4), List(5,6)))
→ (List(1,2) ::: (List(3,4) ::: List(5,6))
```

Folding lists

Using prefix methods rather than ***infix*** operators (/: OR \:)

```
def sum(xs:List[Int]): Int =
  xs.foldLeft (0) ((b,a) => b + a)

sum(List(2,3,6)) → (((0 + 2) + 3) + 6)

def product(xs:List[Int]): Int =
  xs.foldLeft (1) ((b,a) => b * a)

product(List(2,3,6)) → (((1 * 2) * 3) * 6)
```

Folding lists

```
def efficientReverse[T](xs: List[T]): List[T] =
  xss.foldLeft (List[T]()) ((b,a) => a :: b)

efficientReverse(List(1,2,3,4)) →
  (4 :: (3 :: (2 :: (1 :: List())))) →
   4 ::  3 ::  2 ::  1 :: Nil →
   List (4,3,2,1)
```

Folding lists

`reduceLeft` like `foldLeft` but no start value

first function application is to first two elements of the list, second function application is to result of first application and third element, and so on …

```
val lines = Source.fromFile(myFile).getLines.toList

//find the longest line

val longestLine = lines.reduceLeft
   ((a,b) => if (a.length > b.length) a else b)
```

`reduceRight` works in a similar way

Folding lists

`reduceLeft` like `reduceRight` only produce the same result if the function you are using to combine the elements is associative
How about

```
(a: Int, b: Int) => a - b
```

```
val sub1= List(1,2,3,4) reduceRight  (_ - _)
val sub2 = List(1,2,3,4) reduceLeft  (_ - _)
```

Folding lists

```
length ?

exists ?

Average ?

Get ?

forAll ?
```

length

```
def len(list: List[Any]): Int =
  list.foldLeft(0)((sum,_) => sum + 1)
```

exists

```
def exists[A](list: List[A], item: A): Boolean =
  list.foldLeft(false)(_ || _==item)
```

Average

```
def average(list: List[Double]): Double =
  list.foldLeft(0.0)(_+_) /
list.foldLeft(0.0)((r,c) => r+1)
```