Functional Programming in Scala
Part III
Lists/Pairs/Tuples

Raj Sunderraman

Lists

List is a fundamental data structure in functional programming.
List(x1,…,xn)

Examples:

val fruits = List(“apple”,”banana”,”orange”,”mango”)
val numbers = List(10,20,30)
val empty = List()
val nestedList = List(List(1,2,3),List(4,5),List(5,6,7))

Lists are immutable
Lists are recursive (i.e. nested)

LISP-like list structure: (Linked List with car-cdr) - diagram…

List Type

Lists are homogenous. i.e. elements are of same type.
Type of a list of elements of type T is scala.List[T] or just List[T]

e.g.

```
val fruit: List[String] = List(“apple”,”mango”)
val nestedList: List[List[Int]] = List(List(1,2,3),List(4,5))
val empty: List[Nothing] = List()
```

Constructors

All lists are constructed from
- empty list Nil
- construction operation :: (pronounced cons)
  x :: xs gives a new list with first element x, followed by elements of list xs

e.g.

fruit = “apple” :: (“orange” :: (“pear” :: Nil))
nums = 1 :: (2 :: (3 :: (4 :: Nil)))
empty = Nil

:: is right associative

A :: B :: C is interpreted as A :: (B :: C)

List Operations/Patterns

<u>3 basic operations</u>:

head - the first element of the list
tail - the list composed of all the elements except the first
isEmpty - true if list is empty, false otherwise

fruit.head == “apples”
fruit.tail.head == “oranges”
empty.head == throw new NoSuchElementException(“head of empty list”)

<u>List Patterns</u>

Nil
p :: ps
List(p1,…,pn)

1 :: 2 :: xs   denotes a list whose first 2 elements are 1 and 2 and the rest of the list is xs
x :: Nil denotes a singleton list whose element is x
List(1 :: 2 :: xs) is a list of one element, which is the list 1,2,…..
What can you say about the length of x :: y :: List(xs,ys) :: zs ? >=3

Sorting a List

Insertion sort:

```
def isort(xs: List[Int]): List[Int] = xs match {
  case Nil => List()
  case y :: ys => insert(y, isort(ys))
}

def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case Nil => List(x)
  case y :: ys => if (x <= y) x :: xs
                  else y :: insert(x, ys)
}
```

Time Complexity: O(n^2)

Additional List Methods

xs.length - size of xs
xs.last - last element of xs, exception if xs is empty
xs.init - A list of all but the last element, exception if xs is empty
xs take n - List of first n elements, or xs if list is shorter than n
xs drop n - List of last n elements, or xs if list is shorter than n
xs(n) - or written xs apply n, element at index n

<u>Creating new lists</u>

xs ++ ys - concatenation
xs.reverse -
xs updated (n,x) - update index n with x

<u>Finding elements</u>

xs indexOf x - index of x, -1 if not found
xs contains x - same as (xs indexOf >= 0)

Implementations
first, last, init

```
def first[T](xs: List[T]): T = xs match {
    case Nil => throw error("first of empty list")
    case y :: ys => y
}
```

Time complexity: O(1)

```
def last[T](xs: List[T]): T = xs match {
    case Nil => throw error("last of empty list")
    case List(x) => x
    case y :: ys => last(ys)
}
```

Time complexity of last: O(n)

```
def init[T](xs: List[T]): List[T] = xs match {
    case Nil => throw error("init of empty list")
    case List(x) => Nil
    case y :: ys => y :: init(ys)
}
```

Time complexity of last: O(n)

Implementations: concat, reverse

```
def concat[T](xs: List[T], ys: List[T]): List[T] = xs match {
    case Nil => ys
    case z :: zs => z :: concat(zs,ys)
}
```

Time complexity of last: O(|xs|)

```
def reverse[T](xs: List[T]): List[T] = xs match {
    case Nil => Nil
    case y :: ys => reverse(ys) ++ List(y)
}
```

Time complexity of last: O(n^2)
Can be improved to O(n).

<u>Exercises:</u>

Remove the nth element in a list (if no nth element, return original list)

```
def removeAt[T](xs: List[T], n: Int): List[T] =
removeAt(List(1,2,3,4),2)     //> res3: List[Int] = List(1, 3, 4)
```

Flatten a list structure

```
def flatten(xs: List[Any]): List[Any] =
flatten(List(List(1,2),3,List(4,5))) //> res4: List[Any] = List(1, 2, 3, 4, 5)
```

MergeSort - Pairs/Tuples

```
def merge(xs: List[Int], ys: List[Int]): List[Int] = xs match {
  case Nil=> ys
  case x :: xt => ys match {
                    case Nil => xs
                    case y :: yt => if (x < y) x :: merge(xt,ys)
                                    else y :: merge(xs,yt)
                  }
}

def msort(xs: List[Int]): List[Int] = {
    val n = xs.length/2
    if (n == 0) xs
    else {
      val (first,second) = xs splitAt n // Tuple Data Structure
      merge(msort(first), msort(second))
    }
}

def merge2(xs: List[Int], ys: List[Int]): List[Int] = (xs,ys) match {
    case (Nil,ys) => ys
    case (xs,Nil) => ys
    case (x::xt,y::yt) => if (x < y) x::merge(xt,ys) else y::merge(xs,yt)
}
```

Can also access tuple elements as t._1, t._2, etc.

msort for any type, List[T]

The msort solution works only for a list of Int. How to make it more general?

def msort[T](xs: List[T]): List[T] = …

This will not work because of the < comparison in merge. Lets send comparison as a parameter into msort/merge.

```
def msort[T](xs: List[T])(lt: (T,T) => Boolean): List[T] = {
  val n = xs.length/2
  if (n == 0) xs
  else {
    def merge(xs: List[T], ys: List[T]): List[T] = (xs,ys) match {
      case (Nil,ys) => ys
      case (xs,Nil) => xs
      case (x::xt,y::yt) => if (lt(x,y)) x::merge(xt,ys) else y::merge(xs,yt)
    }
    val (first,second) = xs splitAt n
    merge(msort(first)(lt), msort(second)(lt))
}

val xs = List(5,4,3,2)
val fruit = List(“oranges”,”apples”,”bananas”)
msort(xs)((x,y) => x < y)
msort(fruit)((x,y) => x.compareTo(y) < 0)
```

Higher Order Functions for Lists

Some patterns in list processing:

- transform each element in a list in a particular way (map)
- retrieve subset of elements from a list (filter)
- combining elements of a list using an operator (fold)

Functional languages provide us higher-order functions to achieve these patterns

Higher Order List Functions
Map

```
def scaleList(xs: List[Double], factor: Double): List[Double] = xs match {
    case Nil => Nil
    case y :: ys => y*factor :: scaleList(ys,factor)
}

scaleList(List(2.3, 4.5, 6.0), 2)
//> res0: List[Double] = List(4.6, 9.0, 12.0)
```

Actually, Scala Lists have a predefined operator, map, that can do this:

```
List(2.3, 4.5, 6.0) map (x=>2*x)
```

The map function may be defined as follows:

```
abstract class List[T] {
…
  def map[U](f: T=>U): List[U] = this match {
    case Nil => this
    case x :: xs => f(x) :: xs.map(f)
  }
…
}
```

Higher Order - Example

```
def squareList(xs: List[Int]): List[Int] = xs match {
    case Nil => Nil
    case y :: ys => y*y :: squareList(ys)
}

def squareList2(xs: List[Int]): List[Int] = xs.map(x=>x*x)

squareList(List(1,3,6))

squareList2(List(1,3,6))
```

Higher Order List Functions
Filter

```
def posElements(xs: List[Int]): List[Int] = xs match {
    case Nil => Nil
    case y :: ys => if (y > 0) y :: posElements(ys) else posElements(ys)
}

posElements(List(-1,1,2,-3,5))
```

Scala Lists have a “filter” function:

```
List(-1,1,2,-3,5) filter (x => x > 0)
```

The filter function may be defined as follows:

```
abstract class List[T] {
…
  def filter(p: T=>Boolean): List[T] = this match {
    case Nil => this
    case x :: xs => if (p(x)) x:: xs.filter(p) else filter(p)
  }
…
}
```

Higher Order List Functions
Variations of Filter

```
xs filterNot p
  same as xs filter (x => !p(x))

xs partition p
  same as (xs filter (x => p(x)), xs filterNot (x => p(x))

xs takeWhile p
  longest prefix of xs such that the elements satisfy p

xs dropWhile p
  remaining list after all leading elements satisfying p
  are dropped

xs span p
  same as (xs takenWhile (x => p(x)), xs dropWhile (x => p(x))
```

pack/encode

```
def pack[T](xs: List[T]): List[List[T]] = xs match {
    case Nil => Nil
    case y :: ys => pack(ys) match {
                case Nil => List(List(y))
                case z :: zs =>
                   if (z contains y) (y :: z) :: zs else List(y) :: z :: zs
              }
}

pack(List("a","a","a","b","c","c","a"))
//> res7: List[List[String]] = List(List(a, a, a), List(b), List(c, c), List(a))


def encode[T](xs: List[T]): List[(T,Int)] =
  pack(xs).map(x => x match {case a::as => (a,(a::as).length)})

encode(List("a","a","a","b","c","c","a"))
//> res8: List[(String, Int)] = List((a,3), (b,1), (c,2), (a,1))
encode(List())
//> res9: List[(Nothing, Int)] = List()
```

Reduction of Lists
Reduce

Combine elements in a list using a given operator.

e.g.

sum(List(x1,…,xn)) = 0 + x1 + … + xn
product(List(x1,…,xn)) = 1 * x1 * …* xn

We could implement this using recursion as follows:

```
def sum(xs: List[Int]): Int = xs match {
  case Nil => 0
  case y :: ys => y + sum(ys)
}
```

Scala provides an operator, reduceLeft, to do this:

```
def sum(xs: List[Int]): Int = (0 :: xs) reduceLeft ((x,y) => x + y)
def product(xs: List[Int]): Int = (1 :: xs) reduceLeft ((x,y) => x * y)
```

Reduction of Lists - Reduce

Combine elements in a list using a given operator.

sum(List(x1,…,xn)) = 0 + x1 + … + xn
product(List(x1,…,xn)) = 1 * x1 * …* xn

We could implement this using recursion as follows:

```
def sum(xs: List[Int]): Int = xs match {
  case Nil => 0
  case y :: ys => y + sum(ys)
}
```

But, Scala provides an operator, <u>reduceLeft</u>, to do this:

```
def sum(xs: List[Int]): Int = (0 :: xs) reduceLeft ((x,y) => x + y)
def product(xs: List[Int]): Int = (1 :: xs) reduceLeft ((x,y) => x * y)
```

Shorter way to write anonymous functions:
(_ * _) is the same as ((x,y) => (x * y))
Every _ represents a new parameter, going from left to right.

```
def sum(xs: List[Int]): Int = (0 :: xs) reduceLeft (_+_)
def product(xs: List[Int]): Int = (1 :: xs) reduceLeft (_*_)
```

Reduction of Lists - foldLeft

foldLeft is similar to reduceLeft, but takes an accumulator, x, as an additional parameter; the accumulator is returned when called with an empty list.

(List(x1,…,xn) foldLeft z)(op) = (…(z op x1) op …) op xn

So, sum and product can be written as:

```
def sum(xs: List[Int]): Int = (xs foldLeft 0)(_+_)
def product(xs: List[Int]): Int = (xs foldLeft 1)(_*_)
```

reduceLeft and foldLeft may be implemented within List class as follows:

```
abstract class List[T] {…
  def reduceLeft(op: (T,T)=>T): T = this match {
    case Nil => throw new Error(“Nil reduceLeft”)
    case x :: xs => (xs foldLeft x)(op)
  }
  def foldLeft[U](z: U)(op: (U,T) => U): U = this match {
    case Nil => z
    case x :: xs => (xs foldLeft op(z,x))(op)
  }
}
```

Reduction of Lists - foldRight and reduceRight

List(x1,…,xn-1,xn) reduceRight op = x1 op (x2 op (…(xn-1 op xn)…)
(List(x1,…,xn) foldRight acc)(op) = x1 op (…(xn op acc)…)

reduceRight and foldRight may be implemented within List class as follows:

```
abstract class List[T] {…
  def reduceLeft(op: (T,T)=>T): T = this match {
    case Nil => throw new Error("Nil reduceRight")
    case x :: Nil => x
    case x :: xs => op(x, xs reduceRight(op))
  }
  def foldRight[U](z: U)(op[: (U,T) => U): U = this match {
    case Nil => z
    case x :: xs => op(x, (xs foldRight z)(op))
  }
}
```

For operators that are associative and commutative, foldLeft and foldRight are equivalent. But is some cases one is more appropriate than the other.
e.g.

```
def concat[T](xs: List[T], ys: List[T]): List[T] = (xs foldRight ys)(_ :: _)
```

reverse list using foldLeft

```
def reverse[T](xs: List[T]): List[T] = (xs foldLeft z?)(op?)
```

Lets try to figure out z? and op? from examples.

```
Nil
= reverse(Nil)
= (Nil foldLeft z?)(op?)
= z?
```

So, z? is Nil

```
List(x)
= reverse(List(x))
= (List(x) foldLeft Nil)(op?)
= op?(Nil,x)
= x :: Nil
```

So, op? is :: with its operands reversed.

```
def reverse[T](xs: List[T]): List[T] =  (xs foldLeft List[T]())((xs, x) => x :: xs )
```

map, length using foldRight

```scala
def mapFun[T,U](xs: List[T], f: T => U): List[U] =
  (xs foldRight List[U]())((x, y) => f(x)::y)

mapFun(List(1,2,3,4,5), (x => x * x): Int =>Int )
//> res0: List[Int] = List(1, 4, 9, 16, 25)

def lengthFun[T](xs: List[T]): Int =
  (xs foldRight 0)((x, y) => y+1)

lengthFun(List(1,2,3,4,5,4,3,2,1))
//> res1: Int = 9
```