INTRUDUCTION IN FUNCTIONAL PROGRAMMING WITH SCALA

ifitwasi@gmal.com
+SergeyLobin

SCALA

```
def qsort(l: List[Int]): List[Int] = l match {
  case List() => List()
  case _      =>
    qsort(l.filter(_ < l.head))
      ++ List(l.head)
      ++ qsort(l.filter(_ > l.head))
}
```

GO

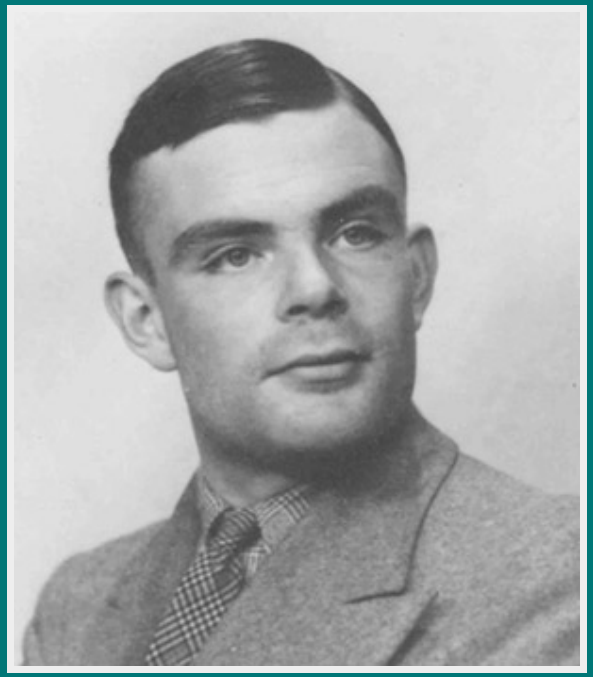
```
func qsort(a []int) []int {
  if len(a) < 2 { return a }
  left, right := 0, len(a) - 1
  pivotIndex := rand.Int() % len(a)
  a[pivotIndex], a[right] = a[right], a[pivotIndex]
  for i := range a {
    if a[i] < a[right] {
      a[i], a[left] = a[left], a[i]
      left++
    }
  }
  a[left], a[right] = a[right], a[left]
  qsort(a[:left])
  qsort(a[left + 1:])
  return a
}
```

AGENDA

- History
- Functions
- Control structures
- Pattern matching
- Functional types
- Higher order functions
- Lazy evaluations

Alonzo Church (1903–1995)

1930 Lambda calculus (wikipedia)



Alan Turing (1912—1954)

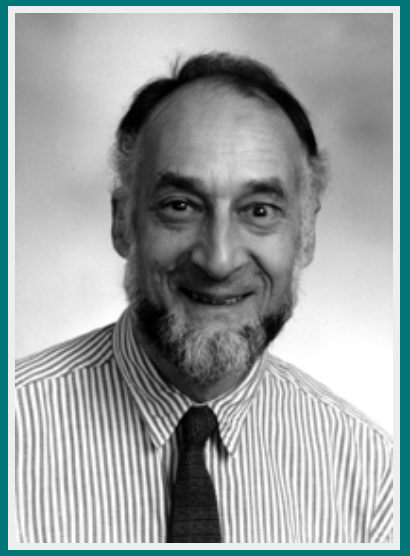
1946 ACE (wikipedia)

John McCarthy (1927—2011)

1958 LISP (2nd ever, 1st functional, garbage collection)

John Backus (1924—2007)

1970s FP (higher-order functions and reasoning)



Robin Milner (1934-2010)

1973 ML (type inference and polymorphic types)

David Turner (1946-)

1985 Miranda (lazy functional language)

1986 Erlang

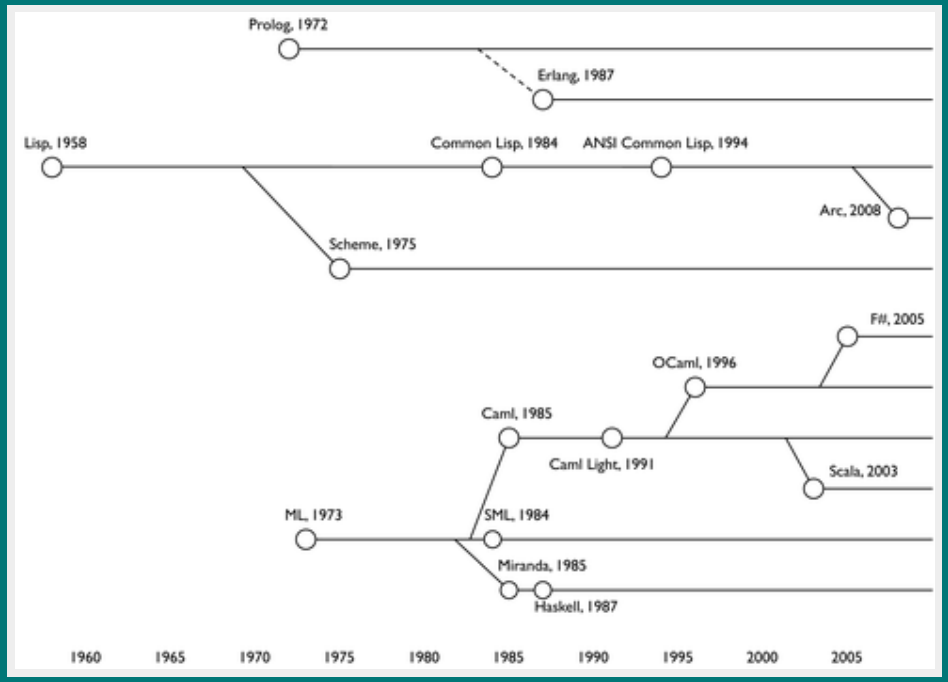
1987 Haskell

2003 Scala

2010 F#

2014 Hack (HHVM)

2014 Java 8


Prolog, 1972
Erlang, 1987
Lisp, 1958
Common Lisp, 1984
ANSI Common Lisp, 1994
Arc, 2008
Scheme, 1975
F#, 2005
OCaml, 1996
Caml, 1985
Caml Light, 1991
Scala, 2003
ML, 1973
SML, 1984
Miranda, 1985
Haskell, 1987
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005

Alonzo Church:

$$\lambda x \rightarrow x+1$$

Simon Marlow:

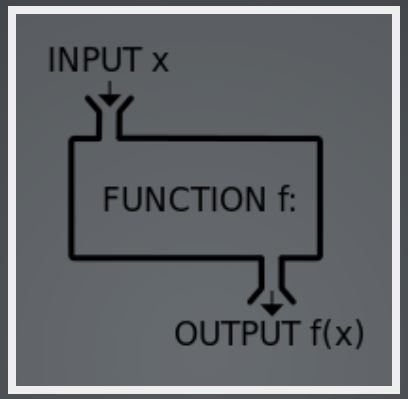
```haskell
\ x -> x + 1 -- haskell
```

Martin Odersky:

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

FUNCTIONS

PURE




```
def adder(a: Int, b: Int): Int = {
  a + b
}

adder(2, 2) // Int = 4
```

COMPOSITION

$$f(g(x)) = (f \circ g)(x)$$

```
def double(x: Int) = x * 2
double(2) // Int = 4

val quad = double _ compose double _ // (double _).compose(double _)
quad(2) // Int = 8
```

CURRING & PARTIAL APPLICATION

```
def a(x: Int)(y: Int) = x + y
a(2)(2) // Int = 4

val plus2 = a(2)_

plus2(2) // Int = 4
```

RECURSION

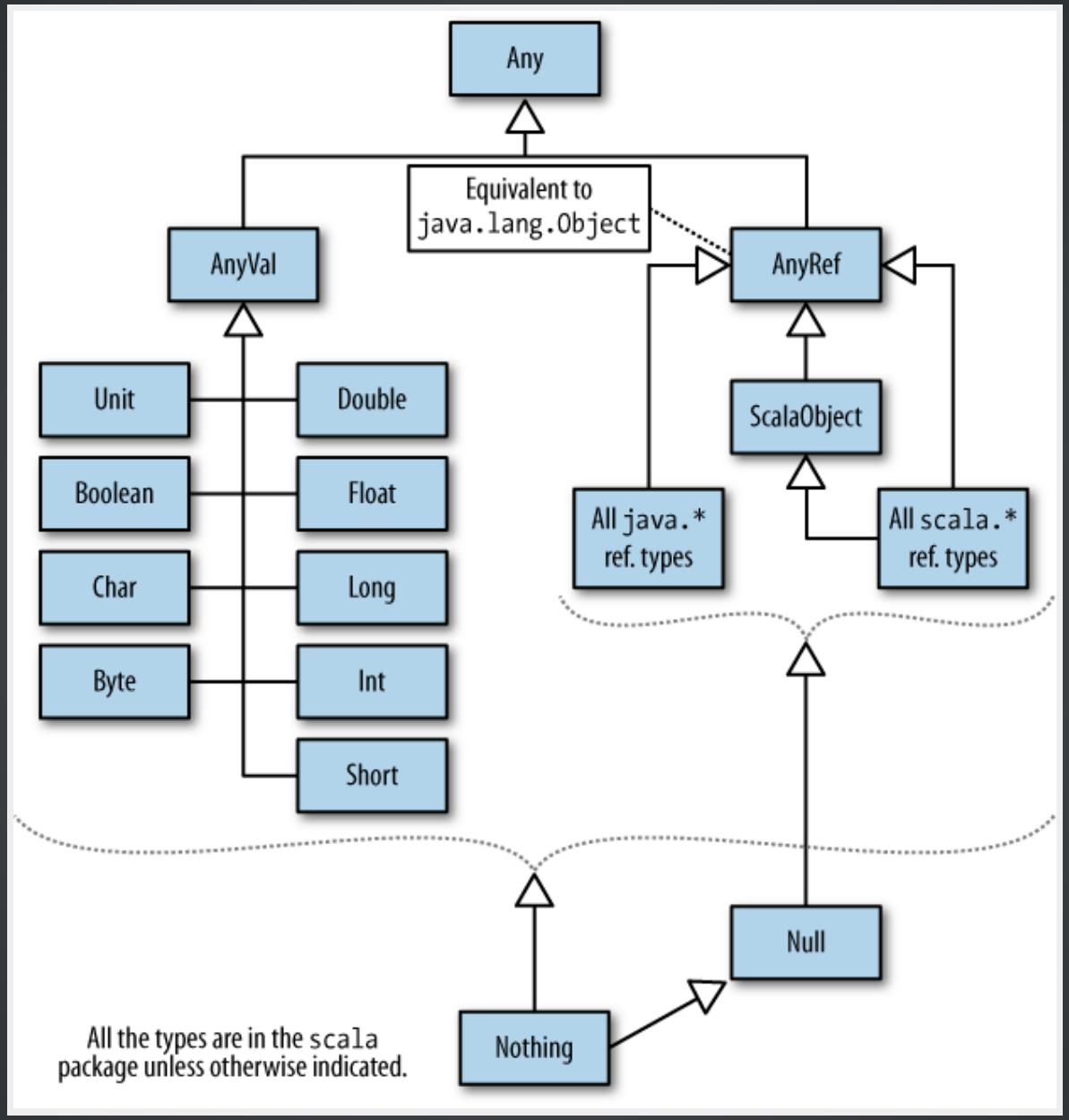
```
def factorial(n: BigInt): BigInt =
  if(n > 0) factorial(n-1) * n else 1

factorial(10)    // 3628800

factorial(10000) // java.lang.StackOverflowError
```

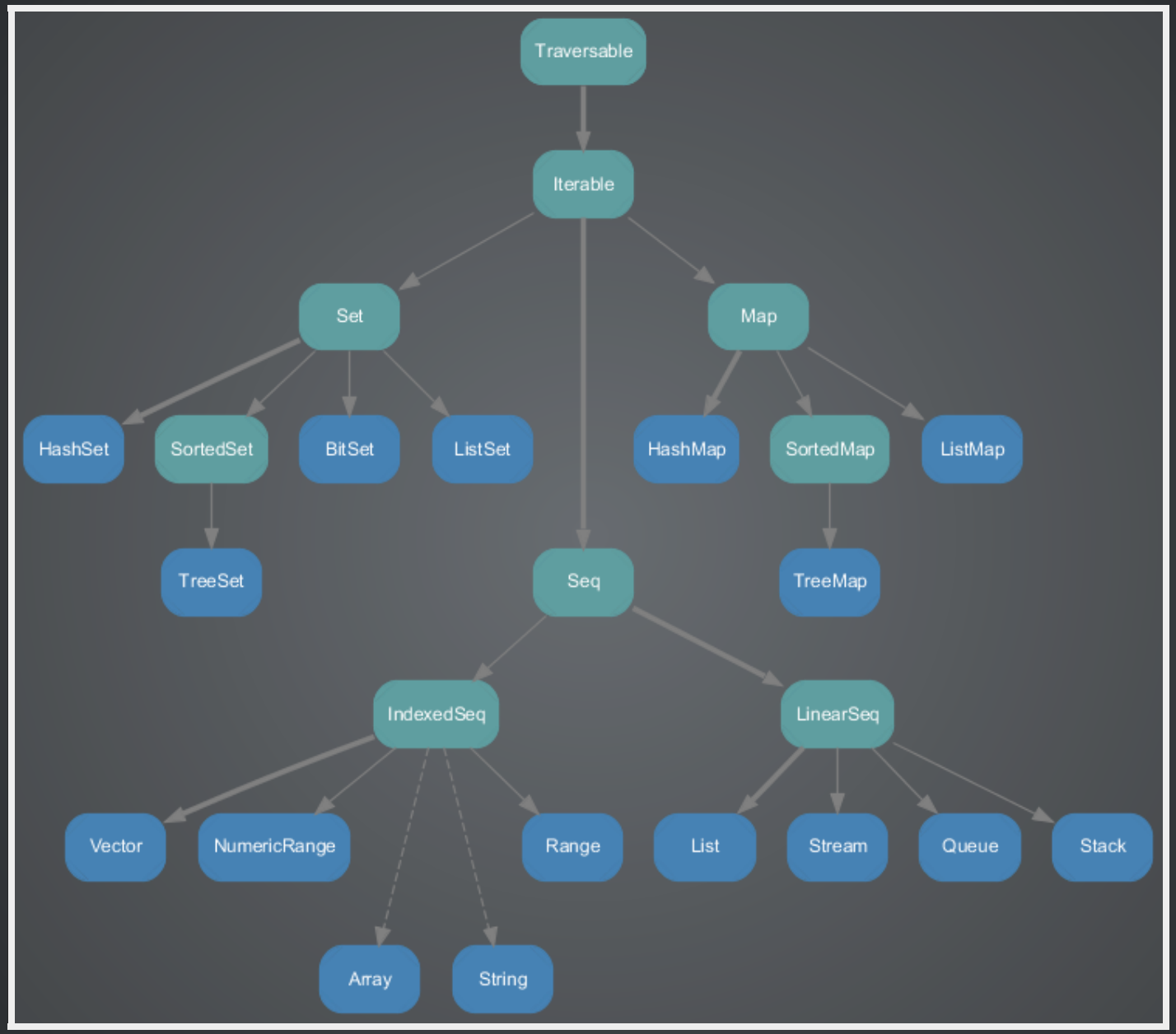
TAIL RECURSION

```
def betterFactorial(x: BigInt) = {
    // auxiliary function
    def loop(n: BigInt, acc: BigInt): BigInt =
      if(n <= 0) acc else loop(n-1, acc*n)

    loop(x, 1)
}

betterFactorial(10000)  // 284625968091705451890641321211986889014805140170279923079417999427441134000376444377299078675778477581588406214231752883004233994015351873905242116138271617481982419982759241828925978789812425312059465996259867065601615720360323979263287367170557419759620994797203461536981198970926112775004841988454104755446424421365733030767036288258035489674611170973695786036701910715127305872810411586405612811653853259684258259955846881464304255898366493170592517172042765974074461334000541940524623034368691540594040662278282483715120383221786446271838229238996389928272218797024593876938030946273322925705554596900278752822425443480211275590191694254290289169072190970836905398737474524833728995218023632827412170402680867692104515558405671725553720152
```

PATTERN MATCHING

```
def bestFactorial(n: BigInt, acc: BigInt = 1): BigInt = n match{
    case 0 => 1
    case n => bestFactorial(n-1, acc*n)
}

bestFactorial(10000)    // 284625968091705451890641321211986889014805140170279923079417999427441134000376444377299078675778477581588406214231752883004233994015351873905242116138271617481982419982759241828925978789812425312059465996259867065601615720360323979263287367170557419759620994797203461536981198970926112775004841988454104755446424421365733030767036288258035489674611170973695786036701910715127305872810411586405612811653853259684258259955846881464304255898366493170592517172042765974074461334000541940524623034368691540594040662278282483715120383221786446271838229238996389928272218797024593876938030946273322925705554596900278752822425443480211275590191694254290289169072190970836905398737474524833728995218023632827412170402680867692104515558405671725553720158521328290342799898184493136106403814 8...
```


Any
Equivalent to
java.lang.Object
AnyVal
AnyRef
Unit
Double
Boolean
Float
Char
Long
Byte
Int
Short
ScalaObject
All java.*
ref. types
All scala.*
ref. types
Null
Nothing
All the types are in the scala
package unless otherwise indicated.


Traversable
Iterable
Set
Map
HashSet
SortedSet
BitSet
ListSet
HashMap
SortedMap
ListMap
TreeSet
Seq
TreeMap
IndexedSeq
LinearSeq
Vector
NumericRange
Range
List
Stream
Queue
Stack
Array
String

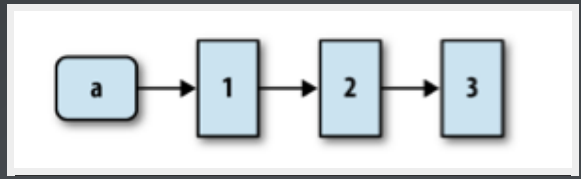
FUNCTIONAL CONTAINERS

```
val l = List(1,2,3)             // l: List[Int] = List(1, 2, 3)

val l = 1 :: 2 :: 3 :: Nil      // l: List[Int] = List(1, 2, 3)

val l = Nil.::(3).::(2).::(1) // l: List[Int] = List(1, 2, 3)

l.head   // Int = 1
l.tail   // List[Int] = List(2, 3)
l.length // Int = 3
```

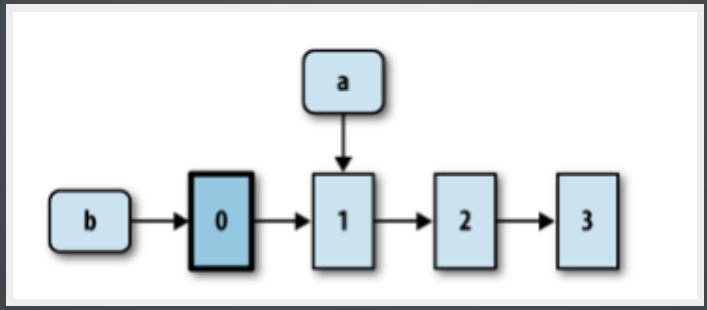
IMMUTABILITY

```
scala> val i = 5
i: Int = 5

scala> i = 6
<console>:8: error: reassignment to val
       i = 6
         ^
```

```scala
val a = List(1, 2, 3) // 1 :: 2 :: 3 :: Nil
```




```scala
val b = 0 :: a  // 0 :: 1 :: 2 :: 3 :: Nil
```



OPERATIONS ON LISTS

```
def myLength[A](l: List[A]): Int = l match {
  case Nil   => 0
  case x::xs => 1 + myLength(xs)
}

myLength(List(1,2,3)) // Int = 3

def filterOdd(l: List[Int]): List[Int] = l match {
  case Nil => List()
  case x::xs if (x % 2 != 0) => x :: filterOdd(xs)
  case x::xs => filterOdd(xs)
}

filterOdd(List(1,2,3)) // List[Int] = List(1, 3)
```

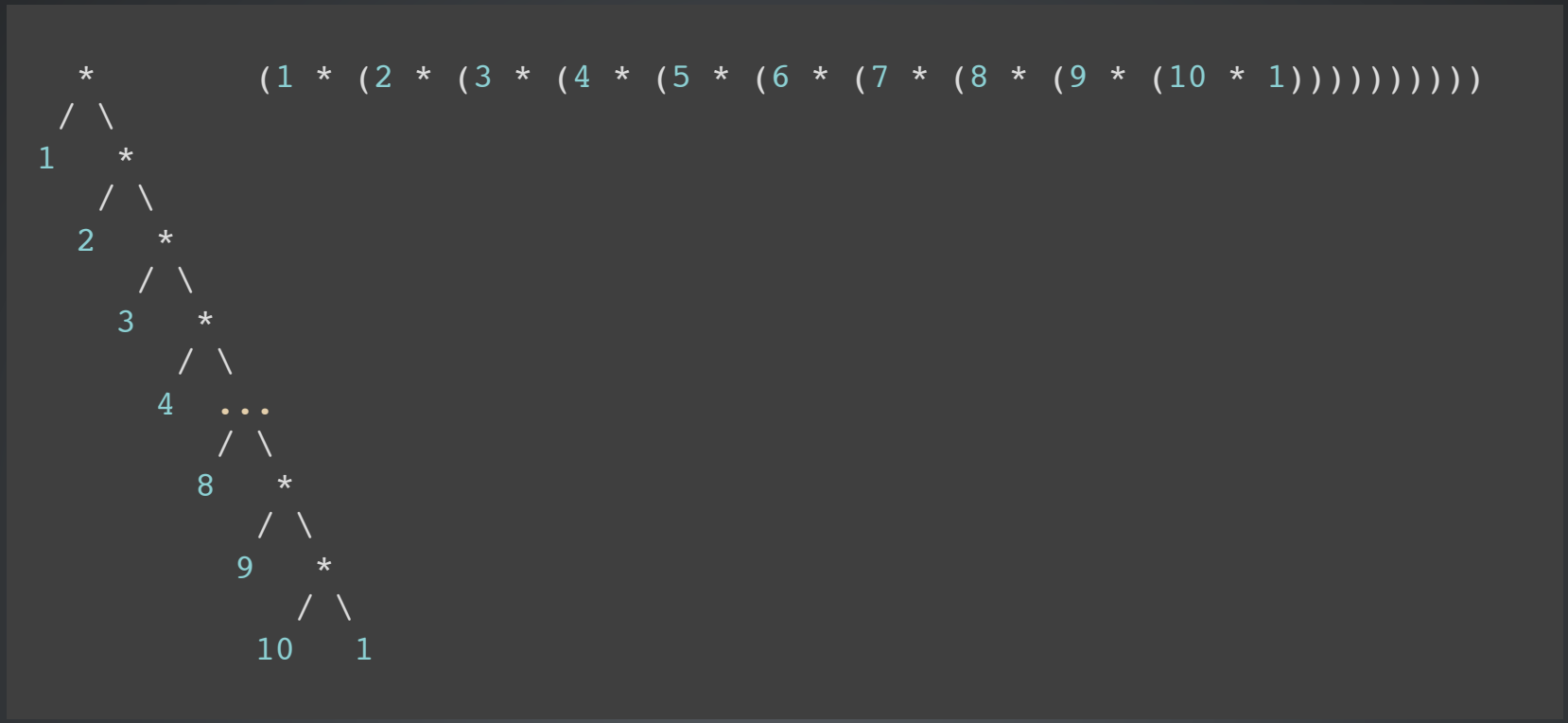
STREAMS

```
val s = Stream.from(0) // Stream(0, ?)
s.toList               // java.lang.OutOfMemoryError: GC overhead limit exceed

s take 10              // Stream(0, ?)
(s take 10).toList     // List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

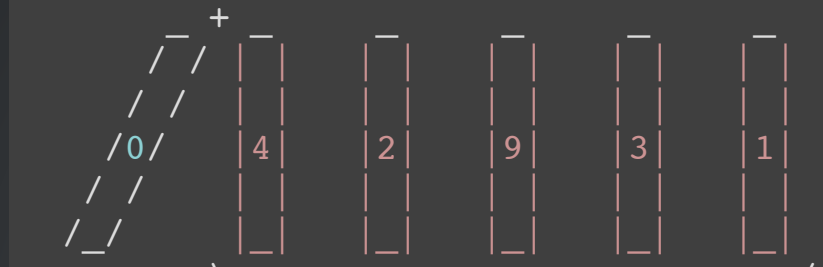
HIGHER ORDER FUNCTIONS

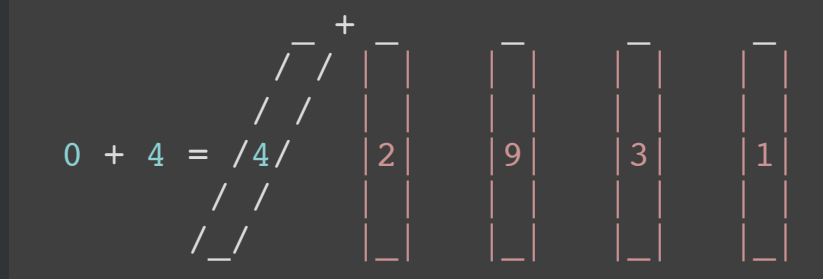
```
// map
val l = List(1, 2, 3)       // List[Int] = List(1, 2, 3)
l.map( x => x * x )         // List[Int] = List(1, 4, 9)

// filter
l.filter( p => p % 2 != 0) // List[Int] = List(1, 3)

1 to 100 takeWhile(_ < 10) // Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

1 to 100 dropWhile(_ <= 90) //Range(91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
```

```
def bestestFactorial(n: BigInt) = (1 to n).foldRight(1)(_*_)

bestestFactorial(10) // Int = 3628800
(1 to 10).foldRight(1)(_*_)
Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).foldRight(1)((x, acc) => x * acc)
Range(1, 2, 3, 4, 5, 6, 7, 8, 9).foldRight(1)( 1 * 10 )
Range(1, 2, 3, 4, 5, 6, 7, 8).foldRight(10)( 10 * 9 )
Range(1, 2, 3, 4, 5, 6, 7).foldRight(90)( 90 * 8 )
...
Range(1).foldRight(1814400)( 1814400 * 2 )
3628800 * 1
```


*
/ \
1 *
/ \
2 *
/ \
3 *
/ \
4 ...
/ \
8 *
/ \
9 *
/ \
10 1
(1 * (2 * (3 * (4 * (5 * (6 * (7 * (8 * (9 * (10 * 1)))))))))

```
val nums = List(4,2,9,3,1)
val sum  = ( 0 /: nums ) (_+_) // nums.foldLeft(0)(_+_)
```





source

```
val m = Map('a' -> 2, 'b' -> 1) // Map[Char,Int] = Map(a -> 2, b -> 1)
m('a') // Int = 2

m('c') // java.util.NoSuchElementException: key not found: c

m.get('c') // Option[Int] = None
m.get('a') // Option[Int] = Some(2)
```

```
def safeGet(m: Map[Char, Int], key: Char): Int = m.get(key) match {
  case None         => -1
  case Some(x: Int) => x
}

safeGet(m, 'a') // Int = 2
safeGet(m, 'c') // Int = -1
```

```
List().tail // java.lang.UnsupportedOperationException: tail of empty list

def safeTail[A](l: List[A]): List[A] = l match {
  case Nil     => List()
  case x :: xs => xs
}

safeTail(List())      // List[Nothing] = List()
safeTail(List(1,2,3)) // List[Int] = List(2, 3)
```

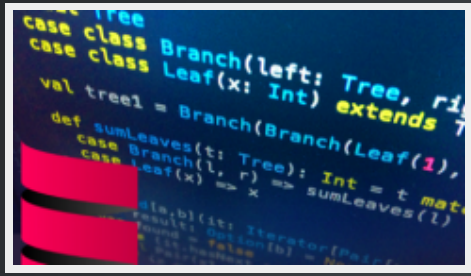
Option & List are Monads

LAZY EVALUATIONS

```
val z = List('a', 'b', 'c') zip List(1, 2, 3)
// List[(Char, Int)] = List((a,1), (b,2), (c,3))

lazy val fibs: Stream[Int] = 0 #:: 1 #::
  fibs.zip(fibs.tail).map { n => n._1 + n._2 } // Stream[Int] = <lazy>

fibs take 10 toList // List[Int] = List(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```

QUESTIONS?

LEARN MORE

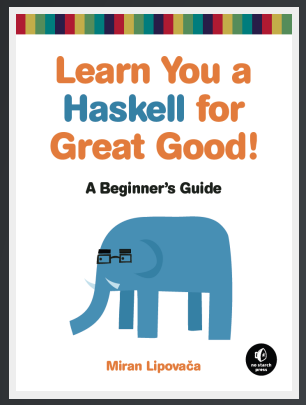


Functional Programming Principles in Scala

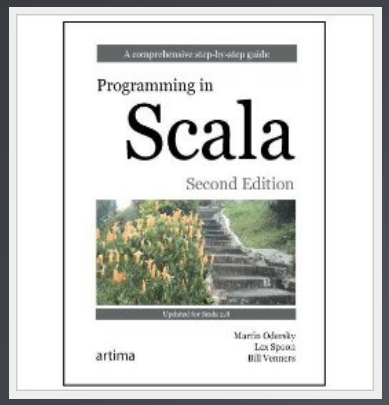
Principles of Reactive Programming

FP101x Introduction to Functional Programming

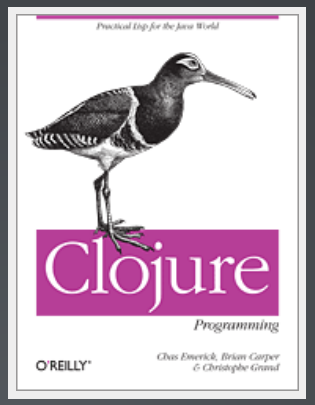
BOOKS



Learn You a Haskell for Great Good!



Programming in Scala



Clojure Programming

О Haskell по-человечески

LINKS

- https://github.com/scalaz/scalaz
- http://www.berniepope.id.au/docs/scala_monads.pdf
- https://github.com/anton-k/ru-neophyte-guide-to-scala
- https://github.com/garu/scala-for-perl5-programmers
- http://fprog.ru/
- http://slides.com/sergeylobin/scala