Higher-order functions

Topics

1. Recursion
2. Tail recursion
3. HOF
4. Polymorphic functions
5. Anonymous function

Topics

1. Recursion
2. Tail recursion
3. HOF
4. Polymorphic functions
5. Anonymous function

The way to write loops functionally is with a recursive function

```
def fib(n: Int): Int = {

  def aux(i: Int, a: Int, b: Int): Int = {
    if (i == n) b
    else aux(i + 1, b, a + b)
  }

  if (n == 1) 0
  else if (n == 2) 1
  else aux(2, 0, 1)
}
```

Topics

1. Recursion
2. Tail recursion
3. HOF
4. Polymorphic functions
5. Anonymous function

A call is said tail position if the caller does nothing that return the value of return call

```scala
def fib(n: Int): Int = {

  @tailrec
  def aux(i: Int, a: Int, b: Int): Int = {
    if (i == n) b
    else aux(i + 1, b, a + b)
  }

  if (n == 1) 0
  else if (n == 2) 1
  else aux(2, 0, 1)

}
```

Topics

1. Recursion
2. Tail recursion
3. HOF
4. Polymorphic functions
5. Anonymous function

Functions can be assigned to variables

Functions can be stored in data structures

Functions can be passed like arguments

HOF are functions that accepts other functions as arguments

```scala
def isSorted[A] (as: Array[A], comp: (A, A) => Boolean) : Boolean = {
  def looping(i: Int) : Boolean = {
    if(i == (as.length - 1)) true
    else
      if (comp(as(i), as(i + 1))) looping(i + 1)
      else false
  }
  looping(0)
}
```

Topics

1. Recursion
2. Tail recursion
3. HOF
4. Polymorphic functions
5. Anonymous function

Functions that works for any type of data

```scala
def isSorted[A] (as: Array[A], comp: (A, A) => Boolean) : Boolean = {
  def looping(i: Int) : Boolean = {
    if(i == (as.length - 1)) true
    else
      if (comp(as(i), as(i + 1))) looping(i + 1)
      else false
  }
  looping(0)
}
```

Topics

1. Recursion
2. Tail recursion
3. HOF
4. Polymorphic functions
5. Anonymous functions

```scala
val f = (x: Int) => x * x
```

```
isSorted(Array(1,2,3,7), (a: Int, b: Int) => a <= b)
```