iscte

TECNOLOGIAS
E ARQUITETURA

Folding
Implicit Parameters
Partially Applied Functions and Currying

Partially based on Martin Odersky slides

Folding

Another common operation is to combine the elements of a list using a given operator.

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

We can implement this with the usual recursive schema:

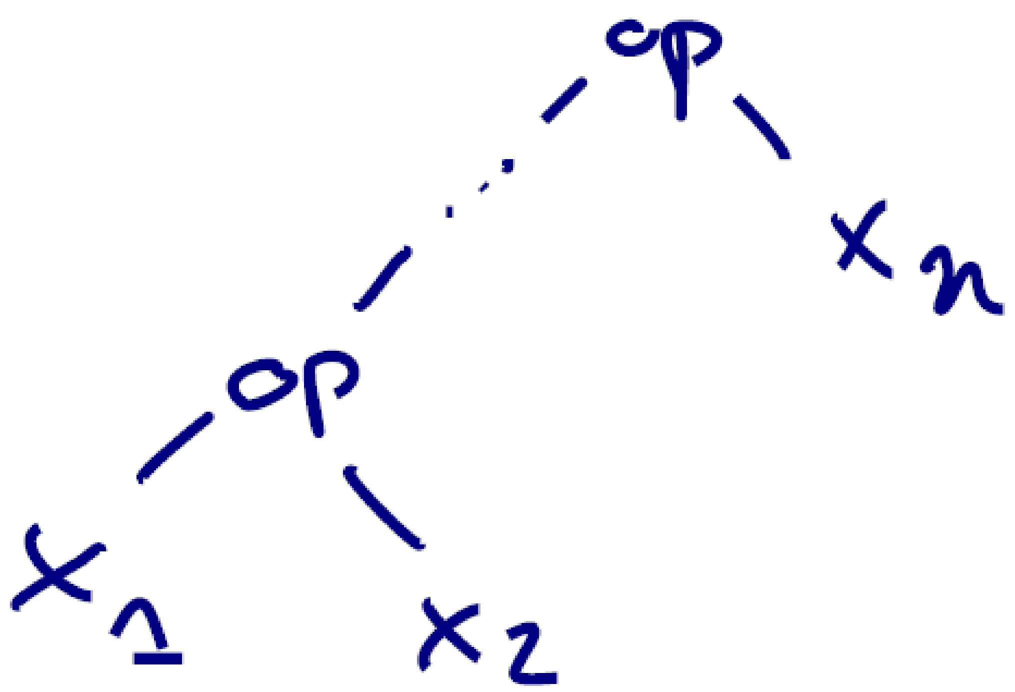
```
def sum(xs: List[Int]): Int = xs match {
  case Nil      => 0
  case y :: ys => y + sum(ys)
}
```

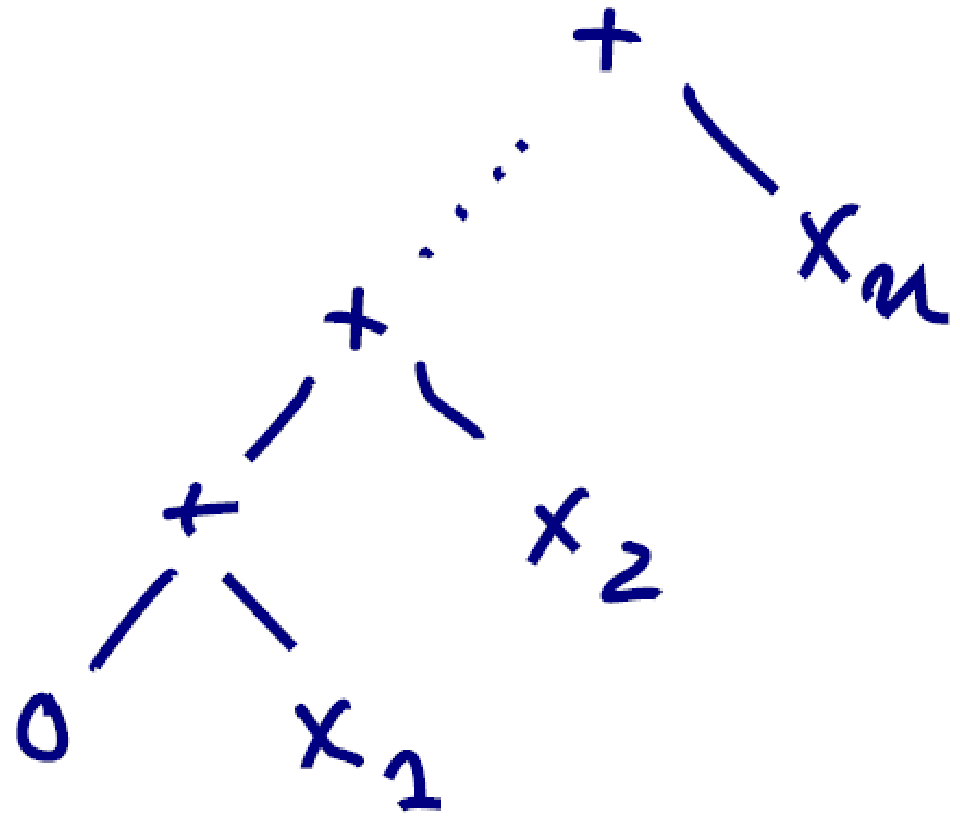
ReduceLeft

This pattern can be abstracted out using the generic method `reduceLeft`:

`reduceLeft` inserts a given binary operator between adjacent elements of a list:

```
List(x1, ..., xn) reduceLeft op   =  (...(x1 op x2) op ... ) op xn
```





Using `reduceLeft`, we can simplify:

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

ReduceLeft – shorter way

Instead of ((x, y) => x * y)) one can write:

(_ * _)

Every _ represents a new parameter, going from left to right.

The parameters are defined at the next outer pair of parentheses (or the whole expression if there are no enclosing parentheses).

So, `sum` and `product` can also be expressed like this:

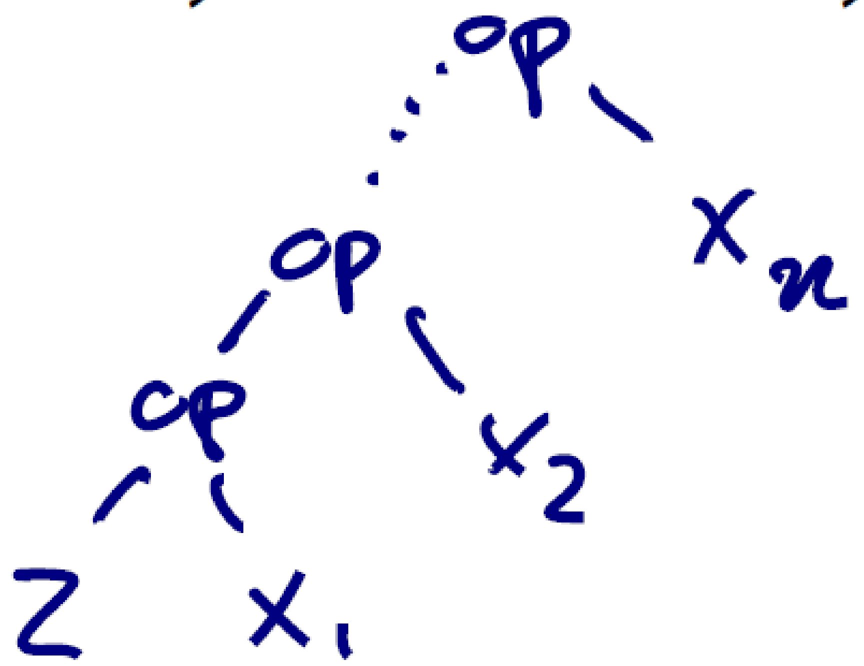
```
def sum(xs: List[Int])     = (0 :: xs) reduceLeft (_ + _)
def product(xs: List[Int]) = (1 :: xs) reduceLeft (_ * _)
```

FoldLeft

The function `reduceLeft` is defined in terms of a more general function, `foldLeft`.

`foldLeft` is like `reduceLeft` but takes an *accumulator*, `z`, as an additional parameter, which is returned when `foldLeft` is called on an empty list.

```
(List(x1, ..., xn) foldLeft z)(op)   =  (...(z op x1) op ... ) op xn
```



So, `sum` and `product` can also be defined as follows:

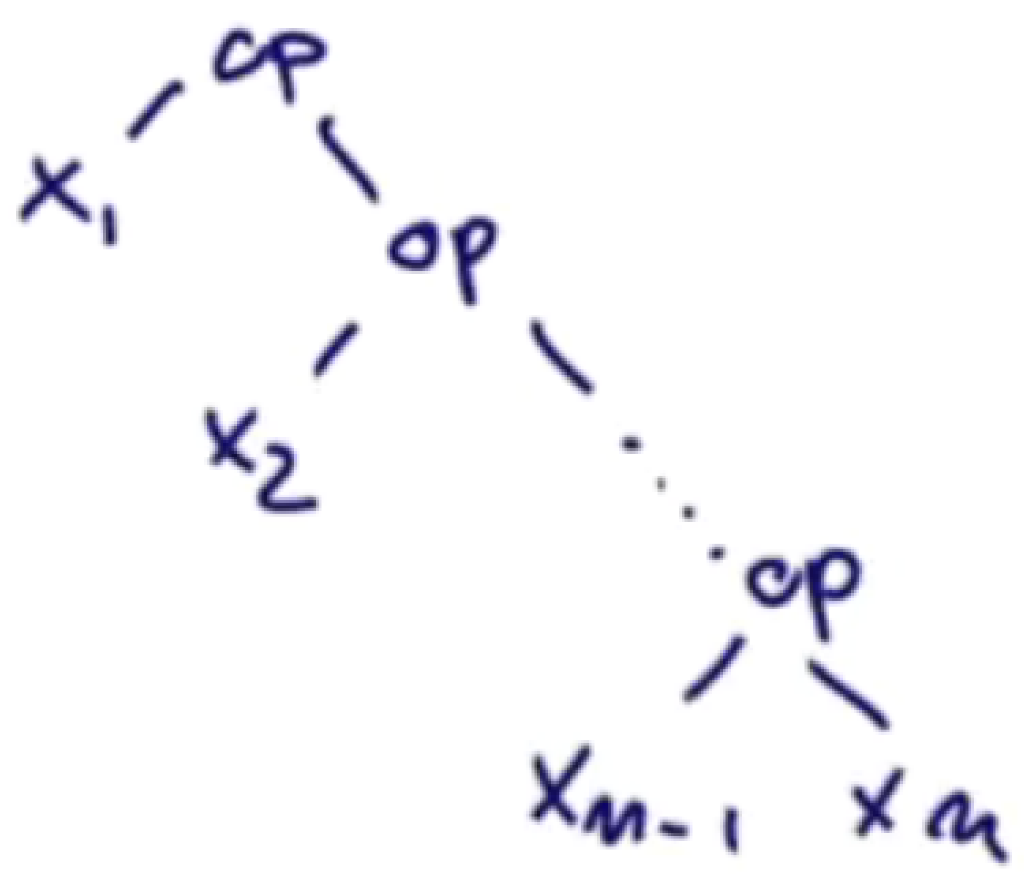
```scala
def sum(xs: List[Int])      = (xs foldLeft 0) (_ + _)
def product(xs: List[Int])  = (xs foldLeft 1) (_ * _)
```

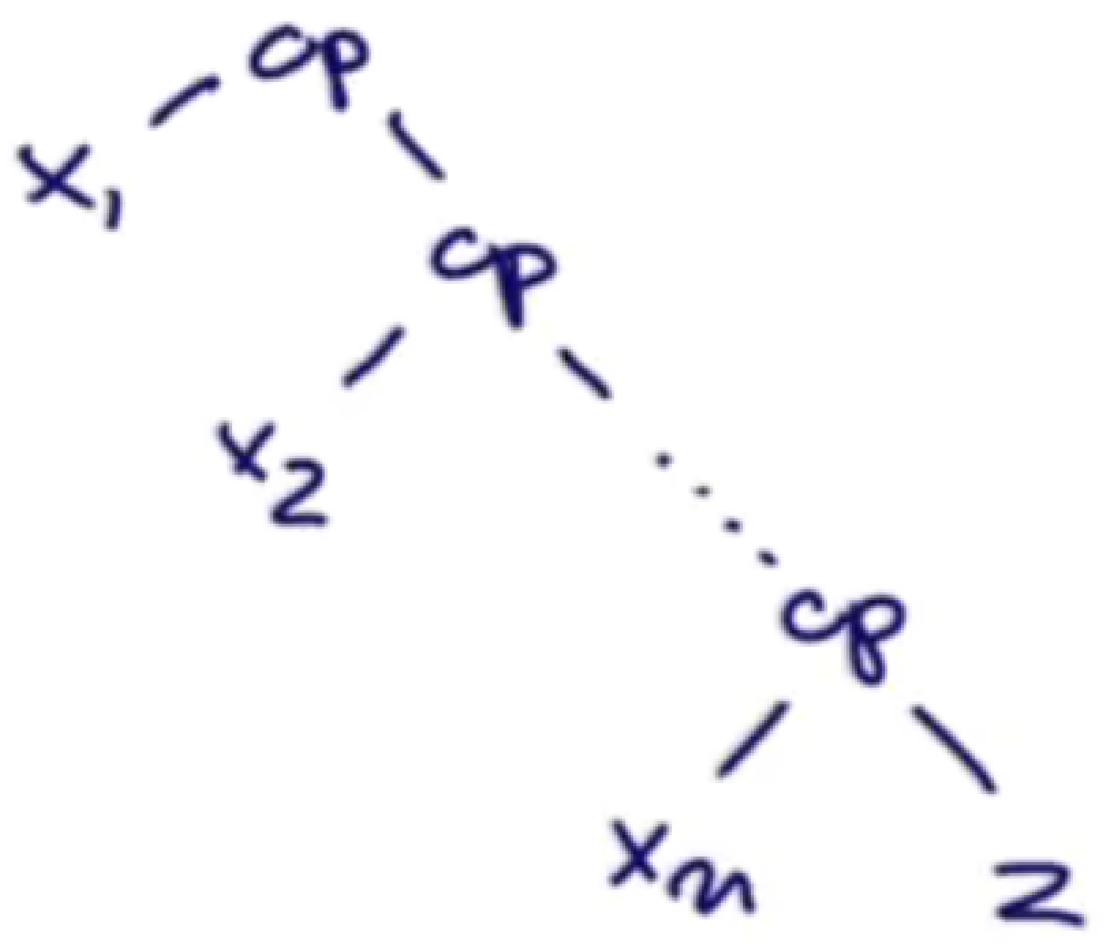
FoldRight and ReduceRight

Applications of `foldLeft` and `reduceLeft` unfold on trees that lean to the left.

They have two dual functions, `foldRight` and `reduceRight`, which produce trees which lean to the right, i.e.,

```
List(x1, ..., x{n-1}, xn) reduceRight op = x1 op ( ... (x{n-1} op xn) ... )
(List(x1, ..., xn) foldRight acc)(op)    = x1 op ( ... (xn op acc) ... )
```





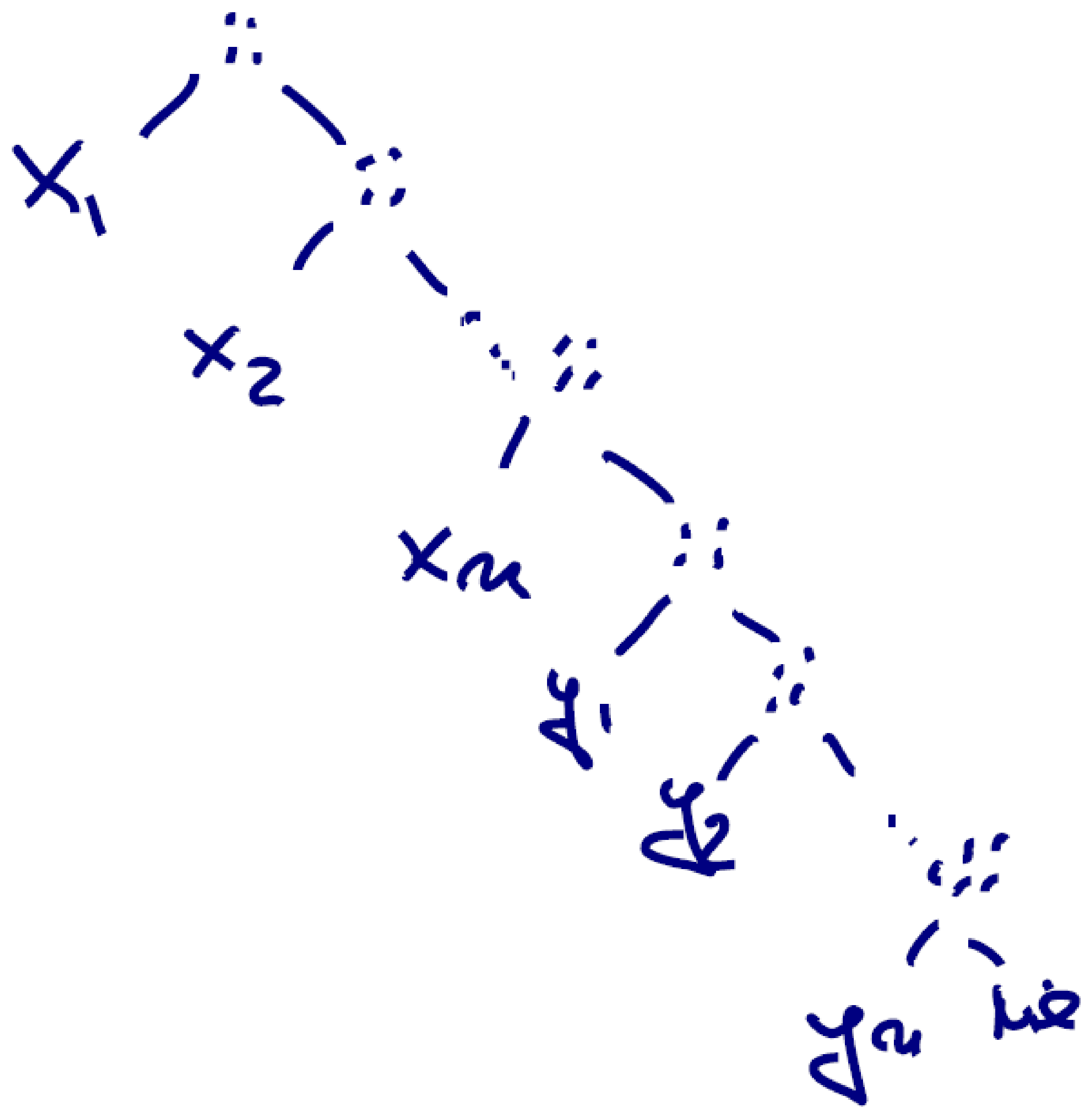
Exercise

Here is another formulation of concat:

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

Here, it isn't possible to replace foldRight by foldLeft. Why?

```
O     The types would not work out
O     The resulting function would not terminate
O     The result would be reversed
```



How to solve it?

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

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

def lengthFun[T](xs: List[T]): Int =
    (xs foldRight 0) (???)
```

and using `foldLeft`?

Implicit Parameters

Sorting Lists – merge sort

```
def mSort(seq: List[Int]): List[Int] = seq match {
  case Nil => Nil
  case xs::Nil => List(xs) //or: case List(xs) => List(xs)
  case _ =>
    val (left, right) = seq splitAt seq.length/2
    merge(mSort(left), mSort(right))
}

def merge(seq1: List[Int], seq2: List[Int]): List[Int] =
   (seq1,seq2) match {
           case (Nil, _) => seq2
           case (_, Nil) => seq1
           case (x::xs, y::ys) =>
            if(x<y) x::merge(xs,seq2)
            else y::merge(seq1,ys)
   }
```

splitAt - returns 2 sublists:
- the elements up the given index,
- the elements from that index

Making Sort more General

Problem: How to parameterize `msort` so that it can also be used for lists with elements other than `Int`?

```
def msort[T](xs: List[T]): List[T] = ...
```

does not work, because the comparison < in `merge` is not defined for arbitrary types `T`.

Idea: Parameterize `merge` with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) = {
  ...
    merge(msort(fst)(lt), msort(snd)(lt))
}
```

Merge then needs to be adapted as follows:

```
def merge(xs: List[T], ys: List[T]) = (xs, ys) match {
  ...
  case (x :: xs1, y :: ys1) =>
    if (lt(x, y)) ...
    else ...
}
```

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val fruit = List("apple", "pear", "orange", "pineapple")

merge(xs)((x: Int, y: Int) => x < y)
merge(fruit)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `merge(xs)`:

```
merge(xs)((x, y) => x < y)
```

Parameterization with Ordered

There is already a class in the standard library that represents orderings.

```
scala.math.Ordering[T]
```

provides ways to compare elements of type `T`. So instead of parameterizing with the `lt` operation directly, we could parameterize with `Ordering` instead:

```
def msort[T](xs: List[T])(ord: Ordering) =

  def merge(xs: List[T], ys: List[T]) =
    ... if (ord.lt(x, y)) ...

  ... merge(msort(fst)(ord), msort(snd)(ord)) ...
```

Ordered Instances

Calling the new `msort` can be done like this:

```
import math.Ordering

msort(nums)(Ordering.Int)
msort(fruits)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

Implicit Parameters

Problem: Passing around `lt` or `ord` values is cumbersome.

We can avoid this by making `ord` an implicit parameter.

```
def msort[T](xs: List[T])(implicit ord: Ordering) =

  def merge(xs: List[T], ys: List[T]) =
    ... if (ord.lt(x, y)) ...

  ... merge(msort(fst), msort(snd)) ...
```

Then calls to `msort` can avoid the ordering parameters:

```
msort(nums)
msort(fruits)
```

The compiler will figure out the right implicit to pass based on the demanded type.

Implicit Parameters (only works with multiple parameter groups)

```
def printIntIfTrue(a: Int)(implicit b: Boolean) = if (b) println(a)

scala> printIntIfTrue(42)(true)
42

scala> printIntIfTrue(1)
<console>:12: error ...

scala> implicit val boo = true
boo: Boolean = true

scala> printIntIfTrue(33)
33
```

Implicit Parameters - Benefits

when refering to a shared resource several times, and you want to keep your code clean.

Example:

- need to reference a database connection several times
- an implicit connection can clean up your code

Rules: in a method or constructor

- only one implicit parameter
- must be the last parameter

Partially Applied Functions (PAF)
Currying

Partially Applied Functions (PAF) and Currying

PAF enable creation of specific functions from the general function

```
def wrap(prefix: String)(html: String)(suffix: String) =
{
   prefix + html + suffix
}

val hello = "Hello, world"
val result = wrap("<div>")(hello)("</div>")

val wrapWithDiv = wrap("<div>")(_: String)("</div>")

scala> wrapWithDiv("<p>Hello, world</p>")
res0: String = <div><p>Hello, world</p></div>
```

PAF - without multiple parameter groups

```
def wrap(prefix: String, html: String, suffix: String) =
{
   prefix + html + suffix
}

val wrapWithDiv = wrap("<div>", _: String, "</div>")
```

or

```
def wrapWithDiv = wrap("<div>", _: String, "</div>")

scala> wrapWithDiv("Hello, world")
res1: String = <div>Hello, world</div>
```

Currying
(only works with multiple parameter groups)

A practical use for **currying** is to specialize functions

```
scala> def multiplier(i: Int)(factor: Int) = i * factor
multiplier: (i: Int)(factor: Int)Int

scala> val byFive = multiplier(5) _
byFive: Int => Int = <function1>

scala> val byTen = multiplier(10) _
byTen: Int => Int = <function1>

scala> byFive(2)
res8: Int = 10
```

```
List(1,2,3) map byTen // ???
```

Currying

```scala
def msort[T](less: (T, T) => Boolean)
    (xs: List[T]): List[T] = {

  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (less(x, y)) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }

  val n = xs.length / 2
  if (n == 0) xs
  else {
    val (ys, zs) = xs splitAt n
    merge(msort(less)(ys), msort(less)(zs))
  }
}
```

Currying

```
scala> msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
res28: List[Int] = List(1, 3, 5, 7)
```

Currying makes it **easy to specialize the function** for particular comparison functions:

```
scala> val intSort = msort((x: Int, y: Int) => x < y) _
intSort: (List[Int]) => List[Int] = <function>

scala> val reverseIntSort = msort((x: Int, y: Int) => x > y) _
reverseIntSort: (List[Int]) => List[Int] = <function>

scala> intSort(List(3,1,9))
scala> reverseSort(List(3,1,9))
```

PAF vs Currying

Currying: a function that takes multiple arguments can be translated into a series of function calls that each take a single argument.
Every function technically has one argument.

PAF: a function that you manually create by supplying fewer parameters than the initial function defines.

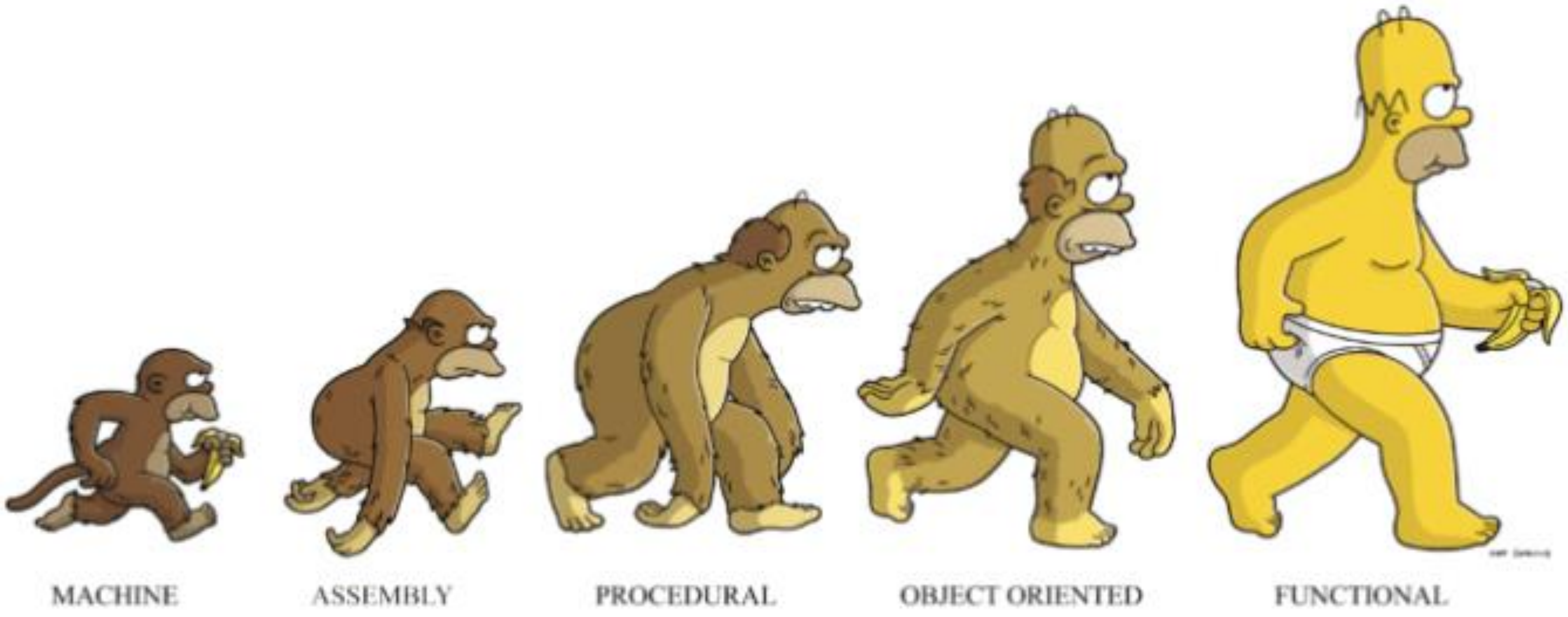
Currying - exercise

Considering:

```
def add(a: Int, b: Int) = a + b
```

How do you specialize the **add** function to become an increment function to be used to increment the elements of a list?

```
List(1,2,3) map ((x) => x+1) // List(2,3,4)
List(1,2,3) map add(1) // List(2,3,4)
List(1,2,3) map add(10) // List(11,12,13)
```


MACHINE
ASSEMBLY
PROCEDURAL
OBJECT ORIENTED
FUNCTIONAL