This Scala notebook uses *BeakerX*, a Two Sigma Open Source project that enhances Jupyter.

http://beakerx.com/ (http://beakerx.com/)

In [1]:
```
scala.util.Properties.versionMsg
```

Out[1]: Scala library version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL

Accumulators

Example 1: Factorial

Definition of Factorial

$$Factorial(n) = \begin{cases} undefined & n < 0 \\ 1 & n < 2 \\ n \times Factorial(n-1) & \text{otherwise} \end{cases}$$

In [2]:
```
def factorial(n: Int): Int = {
    require(n >= 0)
    if (n < 2) 1
    else n * factorial(n-1)
}
```

Out[2]: factorial: (n: Int)Int

In [3]:
```
factorial(8)
```

Out[3]: 40320

We'll leave off the `require` clause from our following definitions for simplicity. Let's see how a call to `factorial(3)` would look if hand-evaluated:

```
factorial(3)
↦ { if (3 < 2) 1 else 3 * factorial(3 - 1) }
↦ { if (false) 1 else 3 * factorial(3 - 1) }
↦ { 3 * factorial(3 - 1) }
↦ { 3 * factorial(2) }
↦ { 3 * { if (2 < 2) 1 else 2 * factorial(2 - 1) } }
↦ { 3 * { if (false) 1 else 2 * factorial(2 - 1) } }
↦ { 3 * { 2 * factorial(2 - 1) } }
↦ { 3 * { 2 * factorial(1) } }
↦ { 3 * { 2 * { if (1 < 2) 1 else 1 * factorial(1 - 1) } } }
↦ { 3 * { 2 * { if (true) 1 else 1 * factorial(1 - 1) } } }
↦ { 3 * { 2 * { 1 } } }
↦ { 3 * { 2 } }
↦ { 6 }
```

In [4]:
```
factorial(3)
```

Out[4]: 6

Imperative-style Factorial

In [5]:
```
def factLoop(n: Int): Int = {
    var acc = 1
    for (i <- 2 to n) {
        acc *= i
    }
    acc
}
```

Out[5]: factLoop: (n: Int)Int

In [6]:
```
factLoop(8)
```

Out[6]:
```
40320
```

Modeling an imperative loop with recursion

In [7]:
```
def factNoLoop(n: Int): Int = {
    def factHelp(i: Int, acc: Int): Int = {
        if (i <= n) factHelp(i+1, acc * i)
        else acc
    }
    factHelp(2, 1)
}
```

Out[7]:
```
factNoLoop: (n: Int)Int
```

In [8]:
```
factNoLoop(8)
```

Out[8]:
```
40320
```

In [9]:
```
def factAcc(n: Int, i: Int = 2, acc: Int = 1): Int = {
    if (i <= n) factAcc(n, i+1, acc * i)
    else acc
}
```

Out[9]:
```
factAcc: (n: Int, i: Int, acc: Int)Int
```

In [10]:
```
factAcc(8)
```

Out[10]:
```
40320
```

Let's see how a call to `factAcc(3)` would look if hand-evaluated:

```
factAcc(3)
↦ { if (2 <= 3) factAcc(3, 2 + 1, 1 * 2) else 1 }
↦ { if (true) factAcc(3, 2 + 1, 2 * 2) else 1 }
↦ { factAcc(3, 2 + 1, 1 * 2) }
↦ { factAcc(3, 3, 1 * 2) }
↦ { factAcc(3, 3, 2) }
↦ { if (3 <= 3) factAcc(3, 3 + 1, 2 * 3) else 2 }
↦ { if (true) factAcc(3, 3 + 1, 2 * 3) else 2 }
↦ { factAcc(3, 3 + 1, 2 * 3) }
↦ { factAcc(3, 4, 2 * 3) }
↦ { factAcc(3, 4, 6) }
↦ { if (4 <= 3) factAcc(3, 4 + 1, 6 * 4) else 6 }
↦ { if (false) factAcc(3, 4 + 1, 6 * 4) else 6 }
↦ { 6 }
```

In [11]:
```
factAcc(3)
```

Out[11]: 6

Example 2: Reverse

`reverse(List(1,2,3))` should result in `List(3,2,1)`

In [12]:
```
def reverse(xs: List[Int], acc: List[Int] = Nil): List[Int] = xs match {
    case y :: ys => reverse(ys, y :: acc)
    case List() => acc
}
```

Out[12]: reverse: (xs: List[Int], acc: List[Int])List[Int]

In [13]:
```
reverse(List(1,2,3)).toString
```

Out[13]: List(3, 2, 1)

Example 3: Map

Remember our original definition of `map` in the lecture slides? It looked something like this:

In [14]:
```
def map(xs: List[Int], f: Int=>Int): List[Int] =
  xs match {
      case y :: ys => f(y) :: map(ys, f)
      case Nil => Nil
  }
```

Out[14]: map: (xs: List[Int], f: Int => Int)List[Int]

In [15]:
```
map(List(1,2,3), 1 + _).toString
```

Out[15]: List(2, 3, 4)

In [16]:
```
def mapLoop(xs0: List[Int], f: Int=>Int): List[Int] = {
  var xs = xs0
  var acc = List.empty[Int]
  while (xs.nonEmpty) {
      val y :: ys = xs
      acc = f(y) :: acc
      xs = ys
  }
  reverse(acc)
}
```

Out[16]: mapLoop: (xs0: List[Int], f: Int => Int)List[Int]

```
In [17]: mapLoop(List(1,2,3), 1 + _).toString
```

```
Out[17]: List(2, 3, 4)
```

Now let's re-implement it using an accumulator to build up the state:

```
In [18]: def mapAcc(xs: List[Int], f: Int=>Int, acc: List[Int] = Nil): List[Int] =
           xs match {
               case y :: ys => mapAcc(ys, f, f(y) :: acc)
               case Nil => reverse(acc)
           }
```

```
Out[18]: mapAcc: (xs: List[Int], f: Int => Int, acc: List[Int])List[Int]
```

```
In [19]: mapAcc(List(1,2,3), 1 + _).toString
```

```
Out[19]: List(2, 3, 4)
```

Tail Recursion

A *tail-recursive* function is a function where all recursive calls are in *tail position*, which means it's the very last thing done in the function's body expression before the returning the resulting value to the caller.

When hand-evaluating Scala programs, a tail-recursive function's body expression reduces to a single recursive call (in the recursive, non-base case of the function), which causes the entire caller's function body to reduce to the recursive call's function body.

As seen in the examples above, recursive functions that use accumulators tend to naturally follow tail-recursive form. Compare the hand-evaluation examples at the top of this notebook for `factorial(3)` vs `factAcc(3)` to see how the non-tail-recursive `factorial` implementation has to unwind and evaluate additional expressions in its call stack, whereas `factAcc` immediately yields a result from the last recursive call because its function body expression always reduces to just the recursive call in the recursive case.

Tail-call optimization

Since a tail-recursive function reduces to the body of the recursive call in the recursive case, a smart compiler can take advantage of the fact that the caller's local state is no longer needed. Tail-call optimization causes tail calls to reuse or replace the caller's *activation record* (also called a *stack frame*) with the recursive call. In other words, a tail-recursive function with 10 tail calls can execute using the same amount of space on the stack as the same function making a million recursive calls, and returns immediately to the caller with its result without needing to unwind a deep recursive call stack.

The annotation `scala.annotation.tailrec` tells the compiler that the following method definition *must* be tail-call optimized. If the method is not tail-recursive, then the compiler throws an error, thus avoiding `StackOverflowError` caused by recursion at run time. This is similar to the `NonNull` annotation in Java 8, where the compiler can help to avoid `NullPointerException` at run time.

In [20]:
```
@scala.annotation.tailrec
final def reverseTR(xs: List[Int], acc: List[Int] = Nil): List[Int] =
    xs match {
        case y :: ys => reverseTR(ys, y :: acc)
        case List() => acc
    }
```

Out[20]:
```
reverseTR: (xs: List[Int], acc: List[Int])List[Int]
```

In [21]:
```
@scala.annotation.tailrec
final def mapTR(xs: List[Int], f: Int=>Int, acc: List[Int] = Nil): List[Int] =
  xs match {
      case y :: ys => mapTR(ys, f, f(y) :: acc)
      case Nil => reverseTR(acc)
  }
```

Out[21]:
```
mapTR: (xs: List[Int], f: Int => Int, acc: List[Int])List[Int]
```

In [22]:
```
mapTR(List.range(0, 1000000), (x: Int) => x).last
```

Out[22]:
```
999999
```

In [23]:
```
map(List.range(0, 1000000), (x: Int) => x).last
```

```
java.lang.StackOverflowError
  at map(<console>:90)
  at map(<console>:90)
  at map(<console>:90)
  ...
```

Why must tail recursive methods be final?

```
class A {
  def f(x: Int): Int = if (x < 10) x else f(x - 3)
}
```

The method `f` is obviously tail recursive, right? However, since neither `A` nor `f` is final, someone can come along and do this:

```
class B extends A {
  override def f(x: Int): Int = if (x > 0) 2 else f(x - 10) * f(x - 5)
}
```

Now if I have some arbitrary instance of type `A`, I don't actually know if invoking `f` will be tail-call optimized!

More details on tail recursion in Scala

You can read more details about how tail-call optimization works in Scala, including how it actually maps down to Java bytecode, in *Programming in Scala 1st ed, 8.9 Tail recursion*:

https://www.artima.com/pins1ed/functions-and-closures.html#8.9 (https://www.artima.com/pins1ed/functions-and-closures.html#8.9)

Using higher-order functions instead

While tail-recursive functions using accumulators are a powerful tool, they're also complex, which makes understanding and maintaining them more difficult. When possible, we should instead use existing higher-order functions to get the same functionality. For example, we can define the Factorial function in Scala in terms of

```
In [24]: def factFold(n: Int): Int = (2 to n).foldLeft(1)(_*_)
Out[24]: factFold: (n: Int)Int

In [25]: factFold(8)
Out[25]: 40320
```

Note that `foldLeft` itself is tail-call optimized, so processing large sequences won't result in a `StackOverflowError`:

```
In [26]: (1 to 1000000).foldLeft(0)(_+_)
Out[26]: 1784293664
```