Recursion and Iteration

Summing by recursion

- Iteration vs. Recursion
  - What is the more natural way to sum a collection? iteration or recursion?
  - A discrete series is often defined recursively (e.g. Fibonacci sequence). I suggest that our tendency towards iteration is a consequence of the Turing/von Neumann architecture.
  - A more mathematical (and natural) way of defining the sum of a list *xs* of numbers would be:
    - sum(xs) = xs.head + sum(xs.tail)

Sum by recursion

- Here, for example, is how to do it in Scala:

```scala
object Sum extends App {

  def sum(xs: Seq[Int]): Int = xs match {
    case Nil => 0
    case h :: t => h + sum(t)
  }

  val xs = Stream.from(1) take 10
  println(sum(xs.toList))
}
```

Recursively call sum

- But haven't we always been taught not to use recursion in our programs if we can avoid it?
  - What's wrong with recursion?
  - Think about what would happen if instead of 10 numbers, we summed 10 *billion* numbers.

Stack Overflow

- We’d get a stack overflow!
  - That’s really bad news. That’s why we were taught not to use recursion!
- But we can actually avoid using the stack provided that the recursion is *tail* recursive.
  - Suppose that the *last* thing we do in our code is the recursive call itself (that’s called *tail* recursion)? We’d also say that the recursive call is in *tail position.* In that case, there’d be nothing we’d need to store on the stack, right?
  - So, we can “unroll” a tail-recursive call into a kind of iteration.

Sum by recursion (take 2)

- Let’s take another look (again, this is Scala):

```
object Sum extends App {

  def sum(xs: Seq[Int]): Int = xs match {
    case Nil => 0
    case h :: t => h + sum(t)
  }

  val xs = Stream.from(1) take 10
  println(sum(xs.toList))
}
```

Recursively call sum *but the* last *thing we do is to add* h *to the result of the recursive call— “+” is in tail position.*

- How can we make this tail-recursive?
    - Actually, it’s quite easy…

Sum by tail-recursion*

- We create an "inner" method which is tail-recursive
  - Its signature is based on two† things:
    - the current value of the result (and which will be yielded when the recursion terminates, in this case when *work* is *Nil*);
    - the work still to do.
  - Here's our new *sum* (note we use *BigInt* because we no longer have a restriction on the size of *xs*):

```
object Sum extends App {
  def sum(xs: Seq[Int]): BigInt = {
    def inner(result: BigInt, work: Seq[Int]): BigInt = work match {
      case Nil => result
      case h :: t => inner(result+h,t)
    }
    inner(0, xs)
  }
  val xs = Stream.from(1) take 10000000
  println(sum(xs.toList))
}
```

Tail-recursively call inner *which* is *the* last *thing we do.*

* also known as tail call recursion

† three if you're processing something 2-dimensional like a tree

Tail recursion in Java

- The problem is that the *Java* compiler doesn't take care of tail-recursion. You have to do it yourself.
- But it's quite easy:

```
f() = if E then {S; return f()} else return Q
f()
```

  where *E* and *Q* are expressions; and where *S* is a series of statements, is equivalent to:

```
while (E) { S }; return Q
```

Example

- Factorial:

```
public long factorial(int n, long r) = {
      if (n>1) return factorial(n-1, r*n) else return r
      }
      factorial(n, 1L);
```

- is equivalent to:

```
public long factorial(int n) = {
      long r = 1;
      while (n>1) {r = r*n; n = n-1}
      return r;
      }
      factorial(n);
```

Multi-way recursion

- The examples we just looked at (sum and factorial) only involve one recursive call. What happens when you have more than one recursive call (e.g. in mergeSort of quickSort)?

- Well, you can never have tail recursion without resorting to an auxiliary ("inner") method (because *both* recursive calls can't be in tail position).

- But, as before, it is quite easy to "unroll" a multi-way recursion into an iteration as we did before.