What can we learn from these examples?

- We have now seen two examples of recursive functions
  - Factorial
  - Sum of digits
- For each example we have seen two versions
  - A version based on a simple mathematical definition
  - A version designed with invariant programming
- The second version has two interesting properties
  - It has *two* arguments; one of the two is an accumulator
  - The recursive call is the last operation in the function body (tail recursion)

The importance of tail recursion

- Let us now take a closer look at why tail recursion is important
- We will do a detailed comparison of the execution of Fact1 and Fact2
  - (This comparison is a first step toward the semantics given in lesson 6)
- We will see why Fact2 (with tail recursion) is more efficient than Fact1 (no tail recursion)
  - Fact1 is based on a simple mathematical definition
  - Fact2 is designed with invariant programming

Comparing Fact1 and Fact2

- Tail recursion is when the recursive call is the last operation in the function body
- N * {Fact1 N-1}   % No tail recursion

  ↑

  After Fact1 is done, we must come back for the multiply.
  Where is the multiplication stored? On a stack!

- {Fact2 I-1 I*A}   % Tail recursion
  The recursive call does not come back!
  All calculations are done *before* Fact2 is called.
  No stack is needed (memory usage is constant).

Stack explosion in Fact1

- 10 * {Fact1 10-1} ⇒
  10 * (9 * {Fact 9-1}) ⇒
  10 * (9 * (8 * {Fact 8-1})) ⇒
  ...
  10 * (9 * (8 * (7 * (6 * (5 * (...(1 * {Fact 0})...) ⇒
  10 * (9 * (8 * (7 * (6 * (5 * (...(1 * 1)...) ⇒
  ...
  3628800

Each line does one computation step

- {Fact2 10-1 10*1} ⇒
  {Fact2 9-1 9*10} ⇒
  {Fact2 8-1 8*90} ⇒
  ...
  {Fact2 1-1 1*3628800}

Comparing functional and imperative loops

- A while loop in the functional paradigm:

```
fun {While S}
    if {IsDone S} then S
    else {While {Transform S}} end /* tail recursion */
end
```

- A while loop in the imperative paradigm:
  (in languages with multiple assignment like Java and C++)

```
state whileLoop(state s) {
    while (!isDone(s))
        s=transform(s); /* assignment */
    return s;
}
```

- In *both* cases, invariant programming is an important design tool