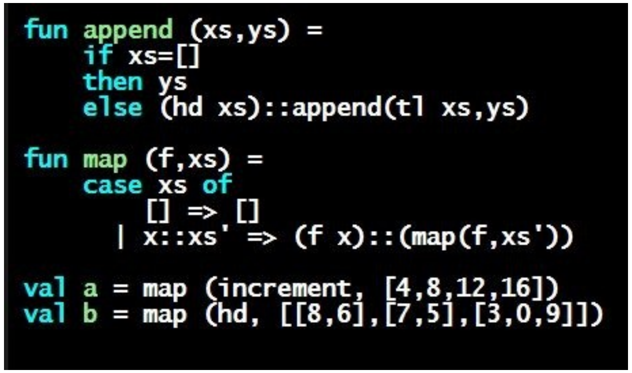


```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman
2013

Tail Recursion

Recursion

Should now be comfortable with recursion:

- No harder than using a loop (whatever that is ☺)
- Often much easier than a loop
  - When processing a tree (e.g., evaluate an arithmetic expression)
  - Examples like appending lists
  - Avoids mutation even for local variables
- Now:
  - How to reason about *efficiency* of recursion
  - The importance of *tail recursion*
  - Using an *accumulator* to achieve tail recursion
  - [No new language features here]

Call-stacks

While a program runs, there is a *call stack* of function calls that have started but not yet returned

- Calling a function `f` pushes an instance of `f` on the stack
- When a call to `f` finishes, it is popped from the stack

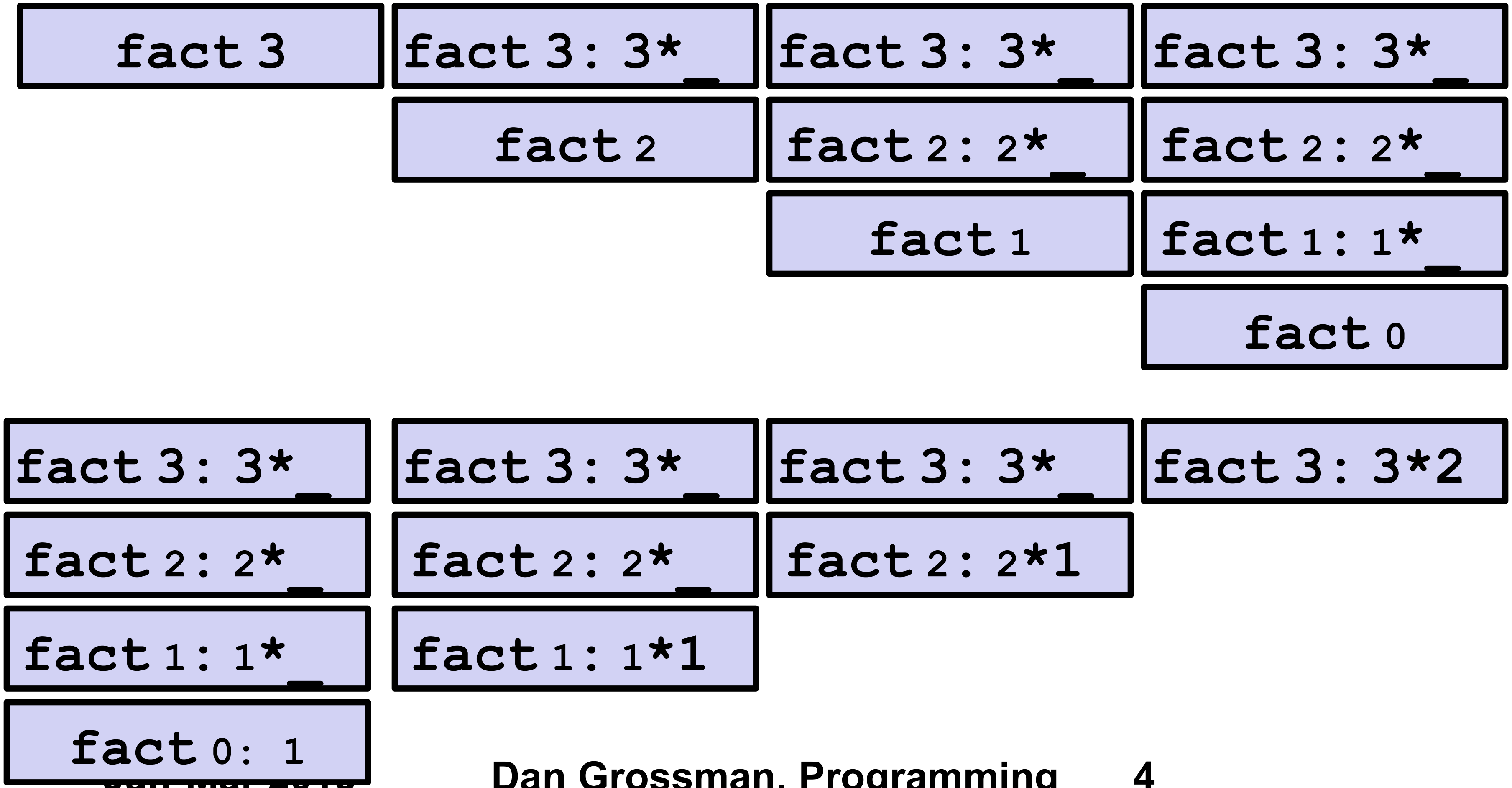
These stack-frames store information like the value of local variables and "what is left to do" in the function

Due to recursion, multiple stack-frames may be calls to the same function

Example

```
fun fact n = if n=0 then 1 else n*fact(n-1)

val x = fact 3
```



fact 3

fact 3: 3*_
fact 2

fact 3: 3*_
fact 2: 2*_
fact 1

fact 3: 3*_
fact 2: 2*_
fact 1: 1*_
fact 0

fact 3: 3*_
fact 2: 2*_
fact 1: 1*_
fact 0: 1

fact 3: 3*_
fact 2: 2*_
fact 1: 1*1

fact 3: 3*_
fact 2: 2*1

fact 3: 3*2

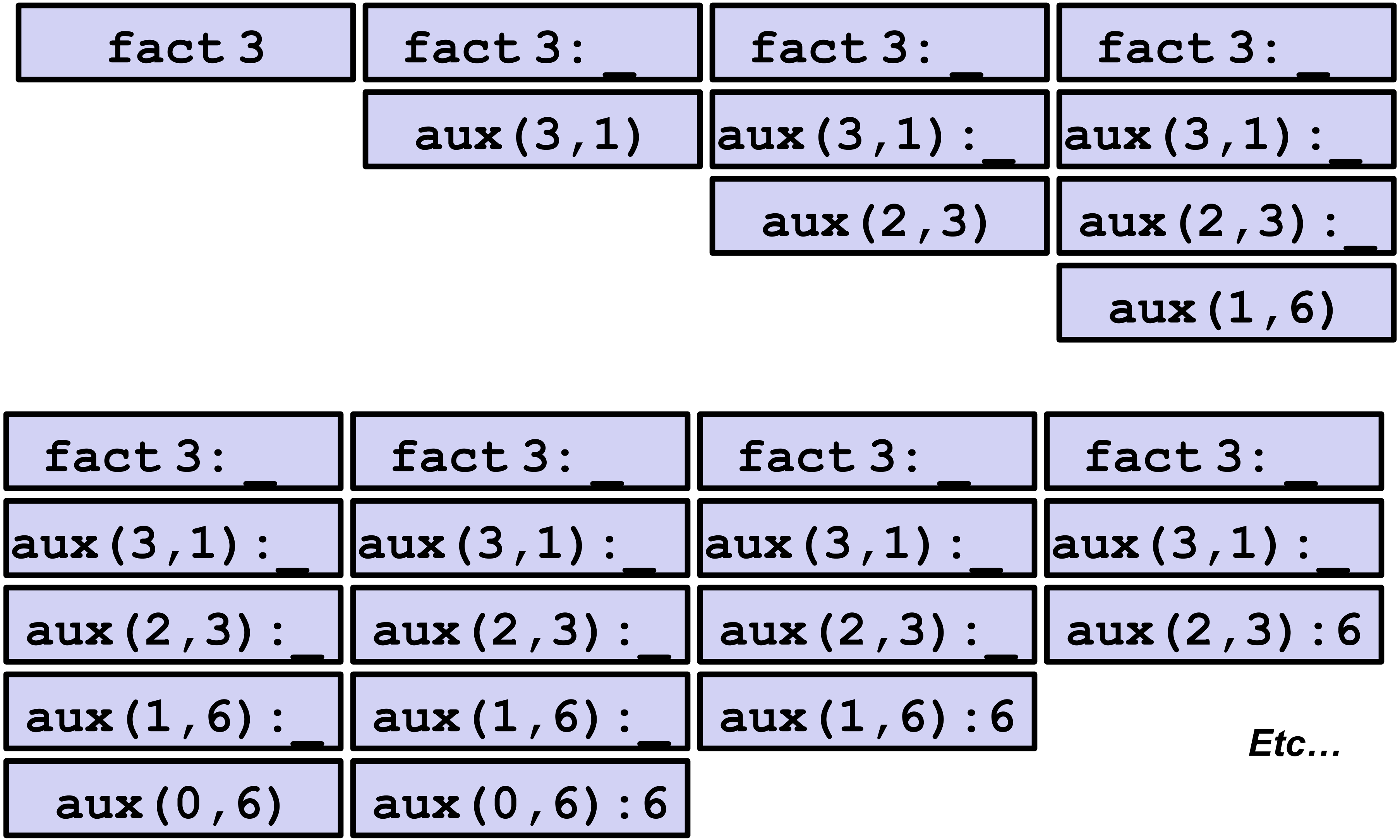
Example Revised

```
fun fact n =
    let fun aux(n,acc) =
           if n=0
           then acc
           else aux(n-1,acc*n)
    in
       aux(n,1)
    end

val x = fact 3
```

Still recursive, more complicated, but the result of recursive calls *is* the result for the caller (no remaining multiplication)

The call-stacks



fact 3

fact 3: _
aux(3,1)

fact 3: _
aux(3,1):_
aux(2,3)

fact 3: _
aux(3,1):_
aux(2,3):_
aux(1,6)

fact 3: _
aux(3,1):_
aux(2,3):_
aux(1,6):_
aux(0,6)

fact 3: _
aux(3,1):_
aux(2,3):_
aux(1,6):_
aux(0,6):6

fact 3: _
aux(3,1):_
aux(2,3):_
aux(1,6):6

fact 3: _
aux(3,1):_
aux(2,3):6

Etc…

An optimization

It is unnecessary to keep around a stack-frame just so it can get a callee's result and return it without any further evaluation

ML recognizes these *tail calls* in the compiler and treats them differently:

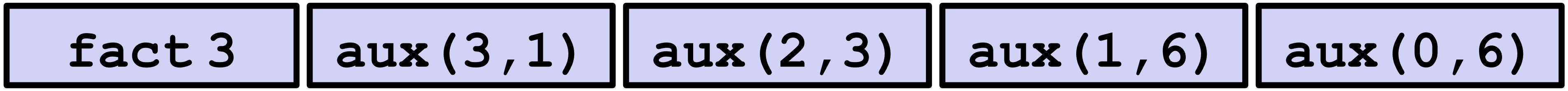
- Pop the caller *before* the call, allowing callee to *reuse* the same stack space
- (Along with other optimizations,) as efficient as a loop

Reasonable to assume all functional-language implementations do tail-call optimization

What really happens

```
fun fact n =
    let fun aux(n,acc) =
           if n=0
           then acc
           else aux(n-1,acc*n)
    in
       aux(n,1)
    end

val x = fact 3
```



fact 3 | aux(3,1) | aux(2,3) | aux(1,6) | aux(0,6)