I WANT TO BUILD A PERFECT HO-SCALE (1/87) MODEL TRAIN LAYOUT OF MY TOWN.

IN YOUR BASEMENT? BAD IDEA. NEVER MAKE A LAYOUT OF THE AREA YOU'RE IN.

WHY NOT?

BECAUSE IT'D INCLUDE A LITTLE 10" REPLICA OF YOUR HOUSE.

SO? THAT'D BE COOL. I'D MAKE TINY REPLICA OF MY ROOMS, MY FURNITURE—

—AND YOUR TRAIN LAYOUT?

21 cm

18 m

2.4 mm

28 mm

320 nm

37 Å

THE MATRYOSHKA LIMIT: IT IS IMPOSSIBLE TO NEST MORE THAN SIX HO LAYOUTS

MY GOD.

YEAH. IT'S THE SECOND RULE OF MODEL TRAIN LAYOUTS: NO NESTING.

... WHAT'S THE FIRST RULE?

"DO NOT TALK ABOUT MODEL TRAIN LAYOUTS" THAT RULE WAS ACTUALLY VOTED IN BY OUR FRIENDS AND FAMILIES. PHILISTINES.
Thinking Inductively

COS 326
David Walker
Princeton University
ANOTHER INDUCTIVE DATA TYPE: THE NATURAL NUMBERS
Natural Numbers

• Natural numbers are a lot like lists
  – both can be defined inductively

• A natural number $n$ is either
  – $0$, or
  – $m + 1$ where $m$ is a smaller natural number

• Functions over naturals $n$ must consider both cases
  – programming the base case $0$ is usually easy
  – programming the inductive case ($m+1$) will often involve recursive calls over smaller numbers

• OCaml doesn't have a built-in type "nat" so we will use "int" instead for now ...
  – “int” has too many values in it (and also not enough)
  – later in the course we could define an abstract type that contains exactly the natural numbers
(* precondition: n is a natural number
return double the input *)

let rec double_nat (n : int) : int =

;;

By definition of naturals:
• n = 0 or
• n = m+1 for some nat m
An Example

(*) precondition: n is a natural number
return double the input *)

let rec double_nat (n : int) : int =
  match n with
  | 0 ->
  | _ ->

By definition of naturals:
• n = 0 or
• n = m+1 for some nat m

two cases:
one for 0
one for m+1
An Example

(* precondition: n is a natural number
return double the input *)

let rec double_nat (n : int) : int =
  match n with
  | 0 -> 0
  | _ ->

solve easy base case first
consider:
what number is double 0?

By definition of naturals:
• n = 0 or
• n = m+1 for some nat m
An Example

(* precondition: n is a natural number return double the input *)

let rec double_nat (n : int) : int =
  match n with
  | 0 -> 0
  | _ -> ????
  ;;

assume double_nat m is correct
where n = m+1
that’s the inductive hypothesis

By definition of naturals:
  • n = 0 or
  • n = m+1 for some nat m
An Example

(* precondition: n is a natural number return double the input *)

let rec double_nat (n : int) : int =
    match n with
    | 0 -> 0
    | _ -> 2 + double_nat (n-1)
;;

assume double_nat m is correct where n = m+1

that’s the inductive hypothesis

By definition of naturals:
• n = 0 or
• n = m+1 for some nat m

I wish I had a pattern (m+1) ... but OCaml doesn’t have it. So I use n-1 to get m.
let double (n : int) : int =

let rec double_nat (n : int) : int =
    match n with
    0 -> 0
    | n -> 2 + double_nat (n-1)
in

if n < 0 then
    failwith "negative input!"
else
    double_nat n

An Example

nest double_nat so it can only be called by double

raises exception

protect precondition of double_nat by wrapping it with dynamic check

later we will see how to create a static guarantee using types
More than one way to decompose naturals

A natural \( n \) is either:
- \( 0 \),
- \( m+1 \), where \( m \) is a natural

A natural \( n \) is either:
- \( 0 \),
- \( 1 \),
- \( m+2 \), where \( m \) is a natural

A natural \( n \) is either:
- \( 0 \),
- \( m \times 2 \)
- \( m \times 2 + 1 \)
A list $xs$ is either:
- $[]$,
- $x :: xs$, where $ys$ is a list

A list $xs$ is either:
- $[]$,
- $[x]$,
- $x :: y :: ys$, where $ys$ is a list

A natural $n$ is either:
- $0$,
- $m * 2$
- $m * 2 + 1$
• Instead of while or for loops, functional programmers use recursive functions

• These functions operate by:
  – decomposing the input data
  – considering all cases
  – some cases are base cases, which do not require recursive calls
  – some cases are inductive cases, which require recursive calls on smaller arguments

• We've seen:
  – lists with cases:
    • (1) empty list, (2) a list with one or more elements
  – natural numbers with cases:
    • (1) zero (2) m+1
  – we'll see many more examples throughout the course