Thinking Inductively

COS 326
David Walker
Princeton University

Administration

- We'll announce on Piazza when you can start an assignment
  - don't start early as there may be changes!
  - sign up for Piazza!
  - Assignment 1 due at 11:59 tonight!

- Program style guide:
  - http://www.cs.princeton.edu/courses/archive/fall13/cos326/style.php

- Read notes:
  - functional basics, type-checking, typed programming
  - thinking recursively (today)

- Extra precept?

Typed Functional Programming

- Functional programs operate by:
  - *extracting information* from their arguments and then
  - *producing new values*

- So far, we've defined *non-recursive* functions in this style to analyze pairs and optional values

- Why? Because *recursive functions typically come from recursive data*
  - Pairs are not recursive -- we need only do a small, (statically) predictable amount of work to get at the information these structures contain
  - Lists and natural numbers can be viewed as recursive
    - not surprisingly, you've defined recursive functions over numbers!

Inductive Programming and Proving

An *inductive data type* T is a datatype defined by:

- a collection of base cases
  - that don't refer to T
- a collection of inductive cases that build new values of type T from pre-existing data of type T

Programming principle:

- solve programming problem for base cases
- solve programming problem for inductive cases by calling function recursively (inductively) on *smaller* data value

Proving principle:

- prove program satisfies property P for base cases
- prove inductive case satisfies property P assuming inductive call on *smaller* data value satisfies property P

LISTS: AN INDUCTIVE DATA TYPE

Lists are Recursive Data

- In O'Caml, a list value is:
  - [ ]   (the empty list)
  - v :: vs   (a value v followed by a shorter list of values vs)

Inductive Case

Base Case

Lists are Inductive Data

- In O'Caml, a list value is:
  - [ ] (the empty list)
  - v :: vs (a value v followed by a shorter list of values vs)

- An example:
  - 2 :: 3 :: 5 :: [ ] has type int list
  - is the same as: 2 :: (3 :: (5 :: [ ]))
  - "::" is called "cons"

- An alternative (better style) syntax:
  - [2; 3; 5]
  - But this is just a shorthand for 2 :: 3 :: 5 :: []. If you ever get confused fall back on the 2 basic primitives: :: and []

Typing Lists

- Typing rules for lists:

(1) [ ] may have any list type t list

(2) if e1 : t and e2 : t list
then e1 :: e2 : t list

Typing Lists

- Typing rules for lists:

(1) [ ] may have any list type t list

(2) if e1 : t and e2 : t list
then e1 :: e2 : t list

- More examples:

(1 + 2) :: (3 + 4) :: [ ] : ??

(2 :: [ ]) :: (5 :: 6 :: [ ]) :: [ ] : ??

[[2]; [5; 6]] : ??

Typing Lists

- Typing rules for lists:

(1) [ ] may have any list type t list

(2) if e1 : t and e2 : t list
then e1 :: e2 : t list

- More examples:

(1 + 2) :: (3 + 4) :: [ ] : int list

(2 :: [ ]) :: (5 :: 6 :: [ ]) :: [ ] : int list list

[[2]; [5; 6]] : int list list

(Remember that the 3rd example is an abbreviation for the 2nd)

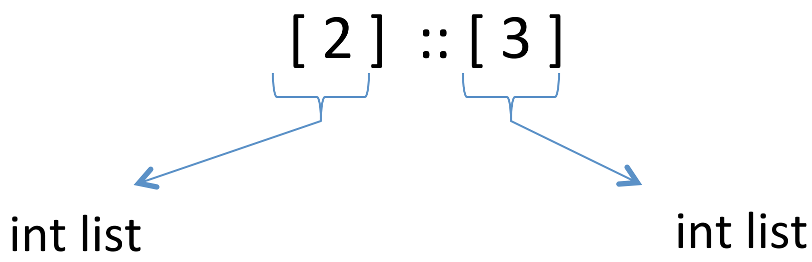
Another Example

- What type does this have?

[2] :: [3]

Another Example

- What type does this have?



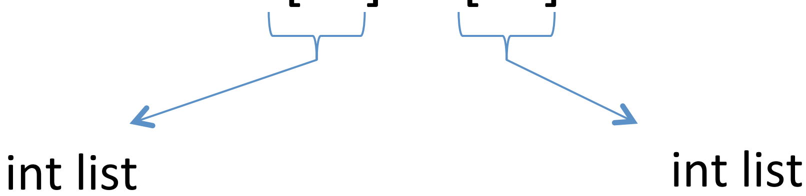
rule: e1 :: e2 : t list if e1 : t and e2 : t list

```
# [2] :: [3];;
Error: This expression has type int but an
         expression was expected of type
         int list
#
```

Another Example

- What type does this have?

[2] :: [3]



- Give me a simple fix that makes the expression type check?

Another Example

- What type does this have?

[2] :: [3]

int list   int list

- Give me a simple fix that makes the expression type check?

Either: 2 :: [3] : int list

Or: [2] :: [[3]] : int list list

Analyzing Lists

- Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

```
(* return Some v, if v is the first list element;
   return None, if the list is empty *)

let head (xs : int list) : int option =



;;
```

Analyzing Lists

- Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

```
(* return Some v, if v is the first list element;
   return None, if the list is empty *)

let head (xs : int list) : int option =
  match xs with
  | [] ->
  | hd :: _ ->
;;
```

we don't care about the contents of the tail of the list so we use the underscore

Analyzing Lists

- Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

```
(* return Some v, if v is the first list element;
   return None, if the list is empty *)

let head (xs : int list) : int option =
  match xs with
  | [] -> None
  | hd :: _ -> Some hd
;;
```

- This function isn't recursive -- we only extracted a small , fixed amount of information from the list -- the first element

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)
```

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =



;;
```

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] ->
  | (x,y) :: tl ->
;;
```

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl ->
;;
```

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> ?? :: ??
;;
```

the result type is int list, so we can speculate that we should create a list

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: ??
;;
```

the first element is the product

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: ??
;;
```

to complete the job, we must compute the products for the rest of the list

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: prods tl
;;
```

Two Parts to Constructing a Function

Think about how to *break down* the input in to cases:

```
let rec prods (xs :
  match xs with

  | [] ->   ...

  | (x,y) :: tl        ...
;;
```

This assumption is called the Induction Hypothesis. You'll use it to prove your program correct.

Assume the recursive call is correct
  (ie: its result satisfies the property you want).
Use its result to *build* correct answer.

```
let rec prods (xs : (int*int) list) : int list =
  ...
  | (x,y) :: tl ->    ...  prods tl ...
```

Recap

Broad steps:

- *break down the input* based on its type in to a set of cases
  - there can be more than one way to do this
- *make the assumption* (the *induction hypothesis*) that your recursive function works correctly when called on a *smaller list*
  - you might have to make 0,1,2 or more recursive calls
- *build the output* (guided by its type) from the results of recursive calls

```
let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: prods tl
;;
```

Another example: zip

```
(* Given two lists of integers,
   return None if the lists are different lengths
   otherwise stitch the lists together to create
     Some of a list of pairs

   zip [2; 3] [4; 5] == Some [(2,4); (3,5)]
   zip [5; 3] [4] == None
   zip [4; 5; 6] [8; 9; 10; 11; 12] == None
*)
```

(Give it a try.)

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =




;;
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') ->
  | (x::xs', []) ->
  | (x::xs', y::ys') ->



;;
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') ->



;;
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') -> (x, y) :: zip xs' ys'



;;
```

is this ok?

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') -> (x, y) :: zip xs' ys'


;;
```

No! zip returns a list option, not a list!
We need to match it and decide if it is Some or None.

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') ->
      (match zip xs' ys' with
         None -> None
       | Some zs -> (x,y) :: zs
;;
```

Closer, but no cigar.

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') ->
      (match zip xs' ys' with
         None -> None
       | Some zs -> Some ((x,y) :: zs)
;;
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | (x::xs', y::ys') ->
      (match zip xs' ys' with
         None -> None
        | Some zs -> Some ((x,y) :: zs))
  | (_, _) -> None
;;
```

Clean up.
Reorganize the cases.
Pattern matching proceeds in order.

A bad list example

```
let rec sum (xs : int list) : int =
  match xs with
  | hd::tl -> hd + sum tl
;;
```

A bad list example

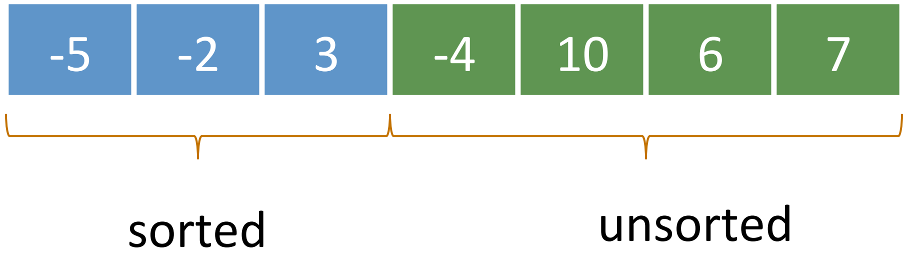
```
let rec sum (xs : int list) : int =
  match xs with
  | hd::tl -> hd + sum tl
;;
```

```
#       Characters 39-78:
  ..match xs with
      hd :: tl -> hd + sum tl..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: []
val sum : int list -> int = <fun>
```

INSERTION SORT

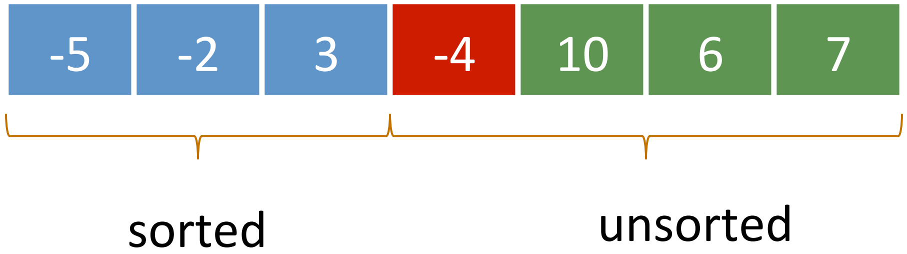
Recall Insertion Sort

- At any point during the insertion sort:
  - some initial segment of the array will be sorted
  - the rest of the array will be in the same (unsorted) order as it was originally

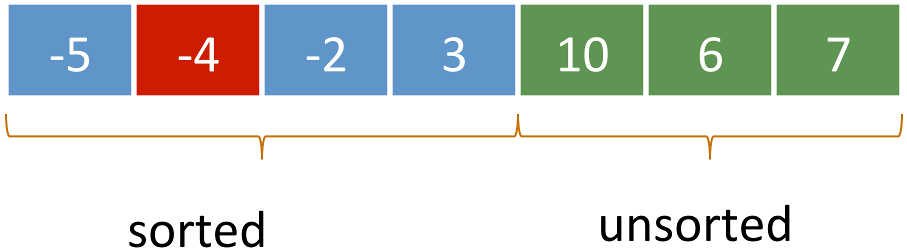


Recall Insertion Sort

- At any point during the insertion sort:
  - some initial segment of the array will be sorted
  - the rest of the array will be in the same (unsorted) order as it was originally

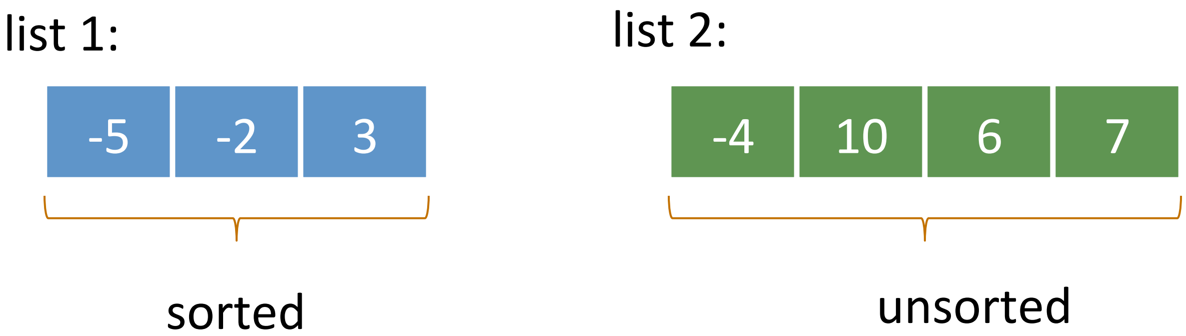


- At each step, take the next item in the array and insert it in order into the sorted portion of the list



Insertion Sort With Lists

- The algorithm is similar, except instead of *one array*, we will maintain *two lists*, a sorted list and an unsorted list



- We'll factor the algorithm:
  - a function to insert in to a sorted list
  - a sorting function that repeatedly inserts

Insert

```
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =





;;
```

Insert

```
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =







;;
```

Insert

```
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =
  match xs with
  | [] ->
  | hd :: tl ->



;;
```

a familiar pattern:
analyze the list by cases

Insert

```
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =
  match xs with
  | [] -> [x]
  | hd :: tl ->




;;
```

insert x in to the empty list

Insert

```
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =
  match xs with
  | [] -> [x]
  | hd :: tl ->
      if hd < x then
        hd :: insert x tl

;;
```

build a new list with:

- hd at the beginning
- the result of inserting x in to the tail of the list afterwards

Insert

```
(* insert x in to sorted list xs *)

let rec insert (x : int) (xs : int list) : int list =
  match xs with
  | [] -> [x]
  | hd :: tl ->
      if hd < x then
        hd :: insert x tl
      else
        x :: xs
;;
```

put x on the front of the list, the rest of the list follows

Insertion Sort

```
type il = int list

insert : int -> il -> il


(* insertion sort *)

let rec insert_sort(xs : il) : il =







;;
```

Insertion Sort

```
type il = int list

insert : int -> il -> il


(* insertion sort *)

let rec insert_sort(xs : il) : il =

  let rec aux (sorted : il) (unsorted : il) : il =



  in


;;
```

Insertion Sort

```
type il = int list

insert : int -> il -> il


(* insertion sort *)

let rec insert_sort(xs : il) : il =

  let rec aux (sorted : il) (unsorted : il) : il =




  in
  aux [] xs

;;
```

Insertion Sort

```
type il = int list

insert : int -> il -> il



(* insertion sort *)

let rec insert_sort(xs : il) : il =

  let rec aux (sorted : il) (unsorted : il) : il =
    match unsorted with
    | [] ->
    | hd :: tl ->
  in
  aux [] xs

;;
```

Insertion Sort

```
type il = int list

insert : int -> il -> il



(* insertion sort *)

let rec insert_sort(xs : il) : il =

  let rec aux (sorted : il) (unsorted : il) : il =
    match unsorted with
    | [] -> sorted
    | hd :: tl -> aux (insert hd sorted) tl
  in
  aux [] xs

;;
```

A COUPLE MORE THOUGHTS ON LISTS

The (Single) List Programming Paradigm

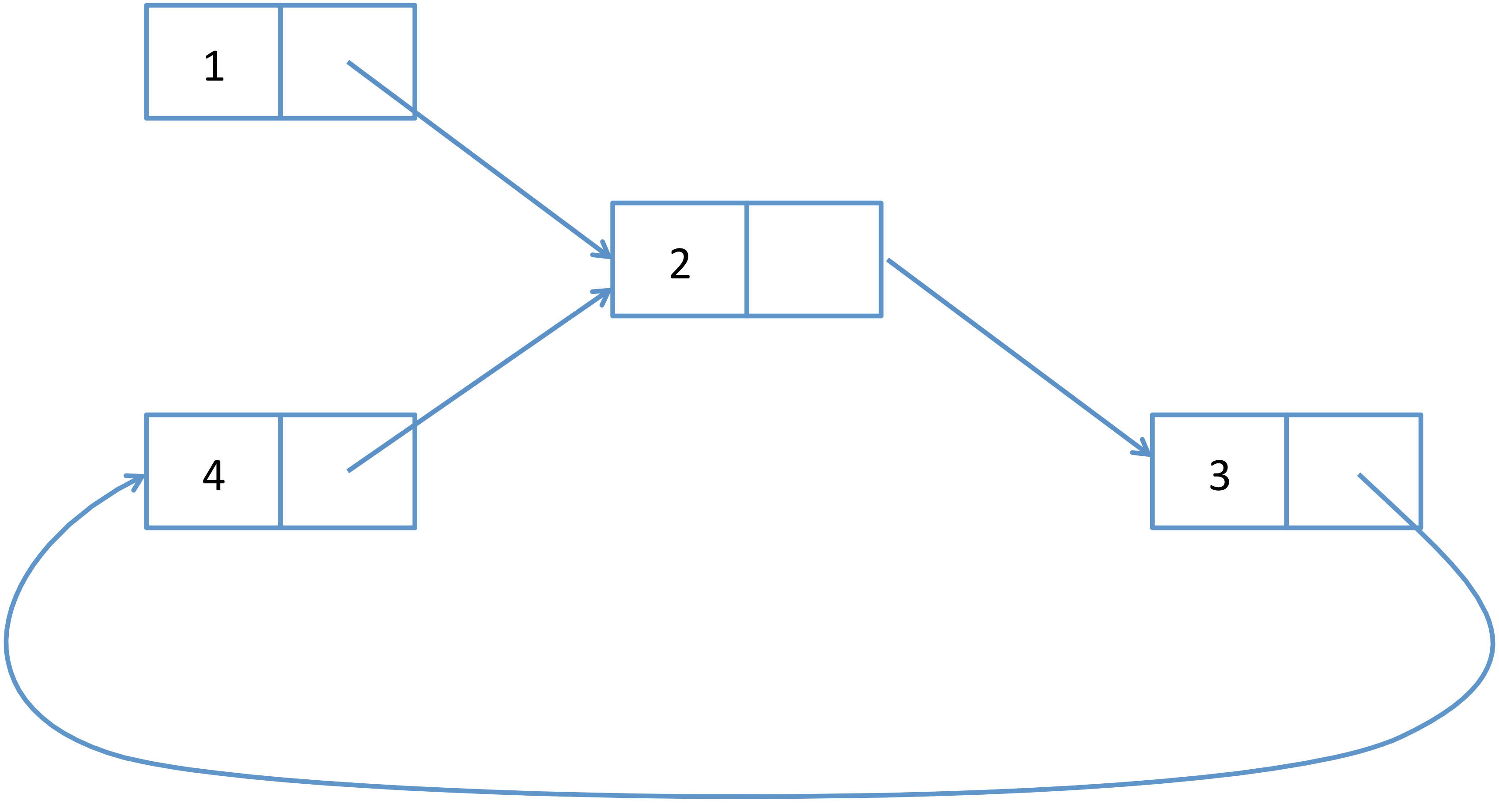
- Recall that a list is either:
  - [ ] (the empty list)
  - v :: vs (a value v followed by a *previously constructed list* vs)

- Some examples:

```
let l0 = [];;                  (* length is 0 *)
let l1 = 1::l0;;               (* length is 1 *)
let l2 = 2::l1;;               (* length is 2 *)
let l3 = 3::l2;;               (* length is 3 *)
…
```

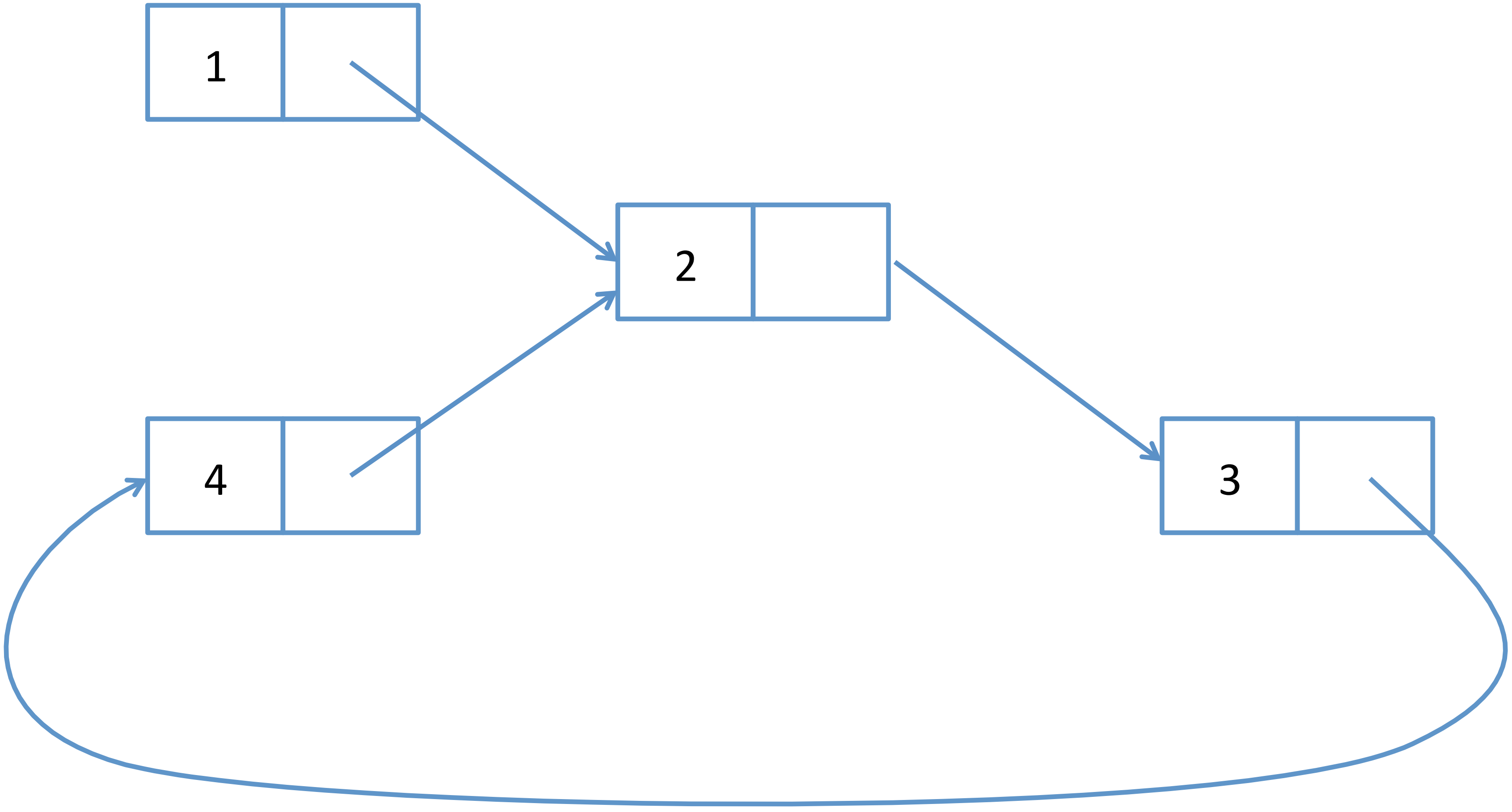
Consider This Picture

- Consider the following picture. How long is the linked structure?
- Can we build a value with type int list to represent it?



Consider This Picture

- How long is it? Infinitely long?
- Can we build a value with type int list to represent it? No!
  - all values with type int list have finite length



The List Type

- Is it a good thing that the type list does not contain any infinitely long lists? Yes!

- A terminating list-processing scheme:

```
let f (xs : int list) : int =
  match xs with
     [] -> … do something not recursive …
   | hd::tail -> …  f tail …
;;
```

terminates because f only called recursively on smaller lists

A Loopy Program

```
let loop (xs : int list) : int =
  match xs with
    [] -> []
  | hd::tail -> hd + loop (0::tail)
;;
```

Does this program terminate?

A Loopy Program

```
let loop (xs : int list) : int =
  match xs with
     [] -> []
   | hd::tail -> hd + loop (0::tail)
;;
```

Does this program terminate? No! Why not? We call loop recursively on (0::tail). This list is the same size as the original list -- not smaller.

Take-home Message

ML has a *strong type system*

- ML *types say a lot* about the set of values that inhabit them

In this case, the tail of the list is *always* shorter than the whole list

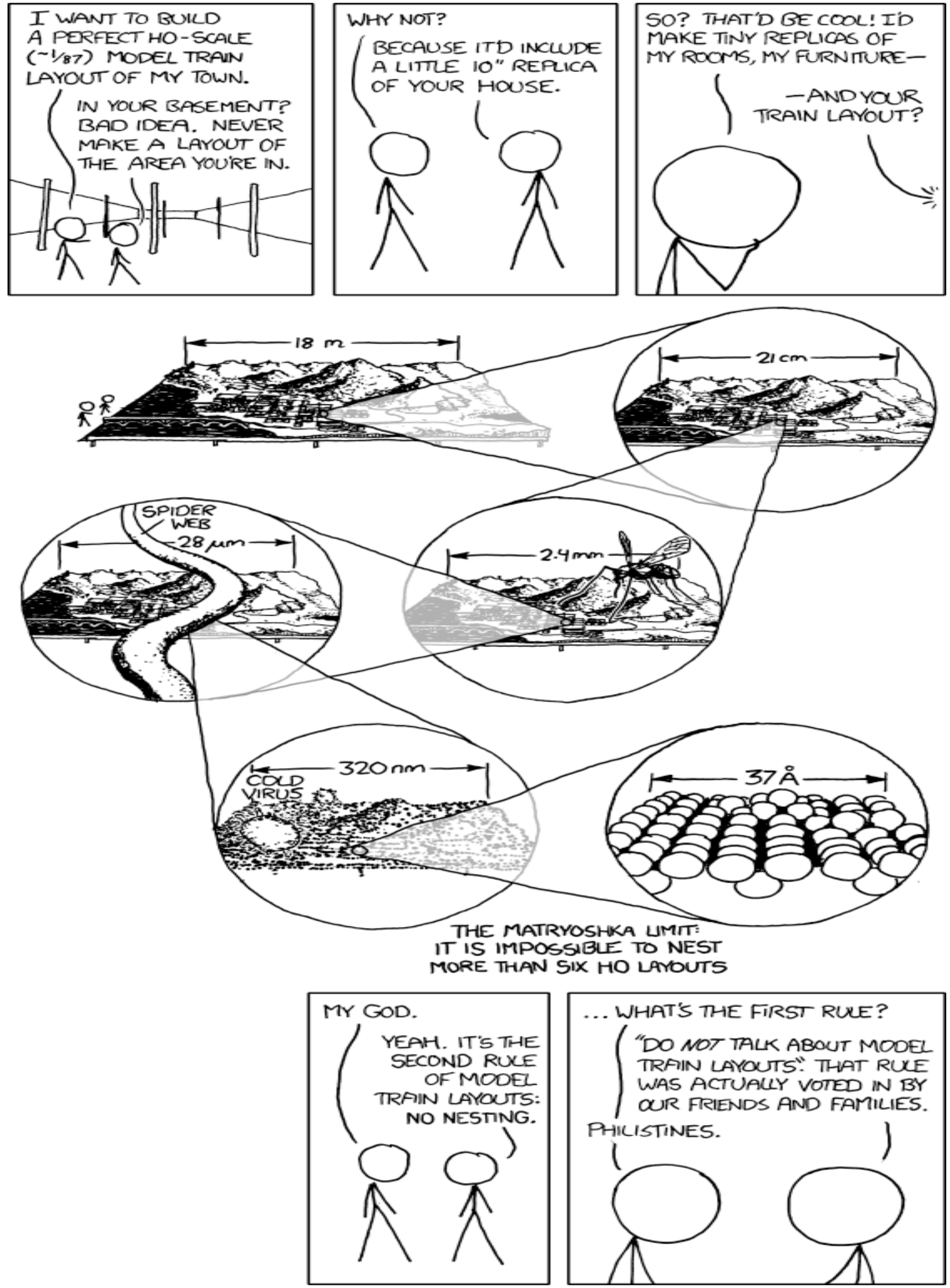
This makes it easy to write functions that terminate; *it would be harder if you had to consider more cases*, such as the case that the tail of a list might loop back on itself. *Moreover OCaml hits you over the head to tell you what the only 2 cases are!*

Note: Just because the list type excludes cyclic structures does not mean that an ML program can't build a cyclic data structure if it wants to. (We'll do that later in the course.)

Rant #2: Imperative lists

- One week from today, ask yourself: Which is easier:
  - Programming with immutable lists in ML?
  - Programming with pointers and mutable lists in C/Java

SCORE: OCAML 2, JAVA 0


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 REPLICAS OF MY ROOMS, MY FURNITURE—
—AND YOUR TRAIN LAYOUT?
18 m
21 cm
SPIDER WEB
28 μm
2.4 mm
320 nm
COLD VIRUS
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.

xkcd

Example problems to practice

- Write a function to sum the elements of a list
  - sum [1; 2; 3] ==> 6
- Write a function to append two lists
  - append [1;2;3] [4;5;6] ==> [1;2;3;4;5;6]
- Write a function to reverse a list
  - rev [1;2;3] ==> [3;2;1]
- Write a function to a list of pairs in to a pair of lists
  - split [(1,2); (3,4); (5,6)] ==> ([1;3;5], [2;4;6])
- Write a function that returns all prefixes of a list
  - prefixes [1;2;3] ==> [[]; [1]; [1;2]; [1;2;3]]

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

An Example

```
(* 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 ->
  | _ ->
;;
```

two cases:
one for 0
one for m+1

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
  | _ ->
;;
```

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.

An Example

```
(* fail if the input is negative
   double the input if it is non-negative *)

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
;;
```

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

unary decomposition

A natural n is either:

- 0,
- 1,
- m+2, where m is a natural

unary even/odd decomposition

A natural n is either:

- 0,
- m*2
- m*2+1

binary decomposition

More than one way to decompose lists

A list xs is either:

- [],
- x::xs, where ys is a list

unary decomposition

A list xs is either:

- [],
- [x],
- x::y::ys, where ys is a list

unary even/odd decomposition

A natural n is either:

- 0,
- m*2
- m*2+1

binary decomposition doesn't
work out as smoothly for lists
as lists have more information content:
they contain structured elements

Summary

- 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

END