Chemical
Biochemical and
Environmental
Engineering

Lean for Scientists and Engineers

Tyler R. Josephson
AI & Theory-Oriented Molecular Science (ATOMS) Lab
University of Maryland, Baltimore County

Technicolor
Madeon

Twitter: @trjosephson
Email: tjo@umbc.edu

Lean for Scientists and Engineers 2024

1. Logic and proofs for scientists and engineers
    1. Introduction to theorem proving
    2. Writing proofs in Lean
    3. Formalizing derivations in science and engineering
2. Functional programming in Lean 4
    1. Functional vs. imperative programming
    2. Numerical vs. symbolic mathematics
    3. Writing executable programs in Lean
3. Provably-correct programs for scientific computing

Schedule (tentative)

Logic and proofs for scientists and engineers
Functional programming in Lean 4
Provably-correct programs for scientific computing

July 9, 2024	Introduction to Lean and proofs
July 10, 2024	Equalities and inequalities
July 16, 2024	Proofs with structure
July 17, 2024	Proofs with structure II
July 23, 2024	Proofs about functions; types
July 24, 2024	Calculus-based-proofs
July 30-31, 2024	Prof. Josephson traveling
August 6, 2024	Functions, recursion, structures
August 7, 2024	Polymorphic functions for floats and reals; lists, arrays
August 13, 2024	Lists, arrays, indexing, and matrices
August 14, 2024	Input / output, compiling Lean to C
August 20, 2024	LeanMD & BET Analysis in Lean
August 21, 2024	SciLean tutorial, by Tomáš Skřivan

Content inspired by:
Mechanics of Proof, by Heather Macbeth
Functional Programming in Lean, by David Christiansen

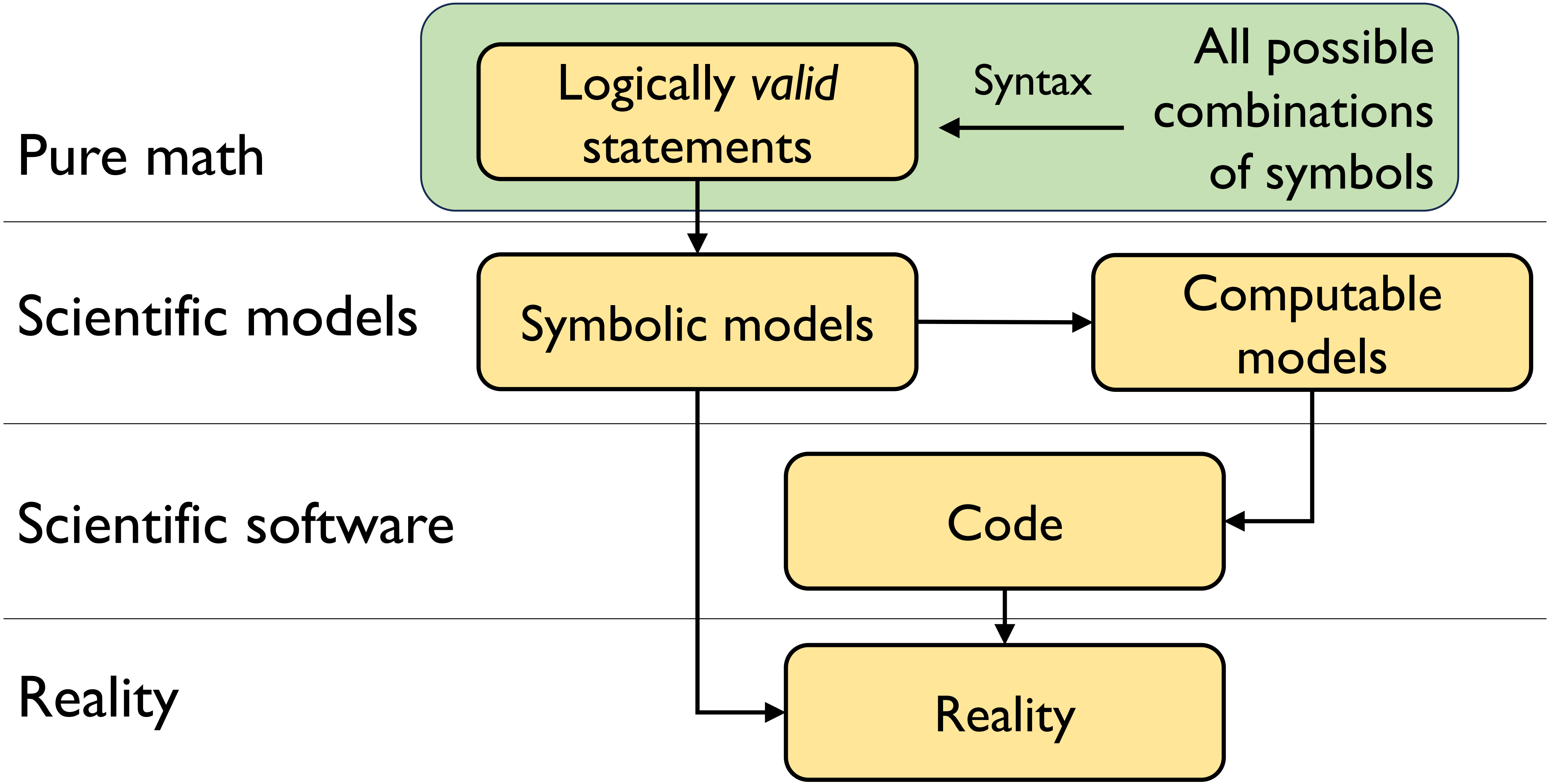
Guest instructor: Tomáš Skřivan

Schedule for today

- Recap Lecture 7
- “Do” notation in Lean
- Polymorphic functions
- Lists and arrays
- Recursion over lists

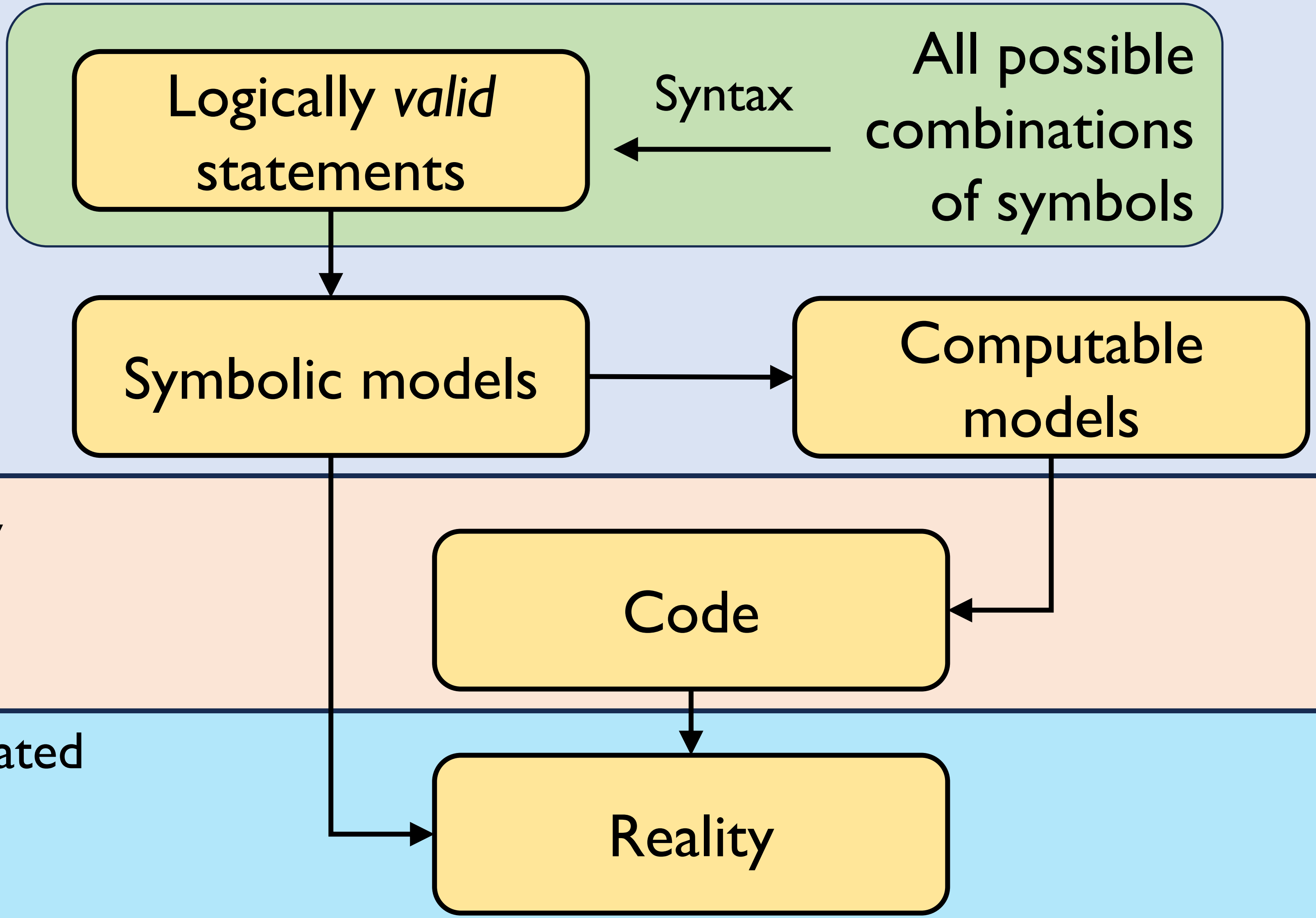
Slide from Lecture 3

Syntax and semantics in scientific computing



Syntax and semantics in scientific computing

Traditionally, the validity of the mathematics and the scientific theory are established **by hand**



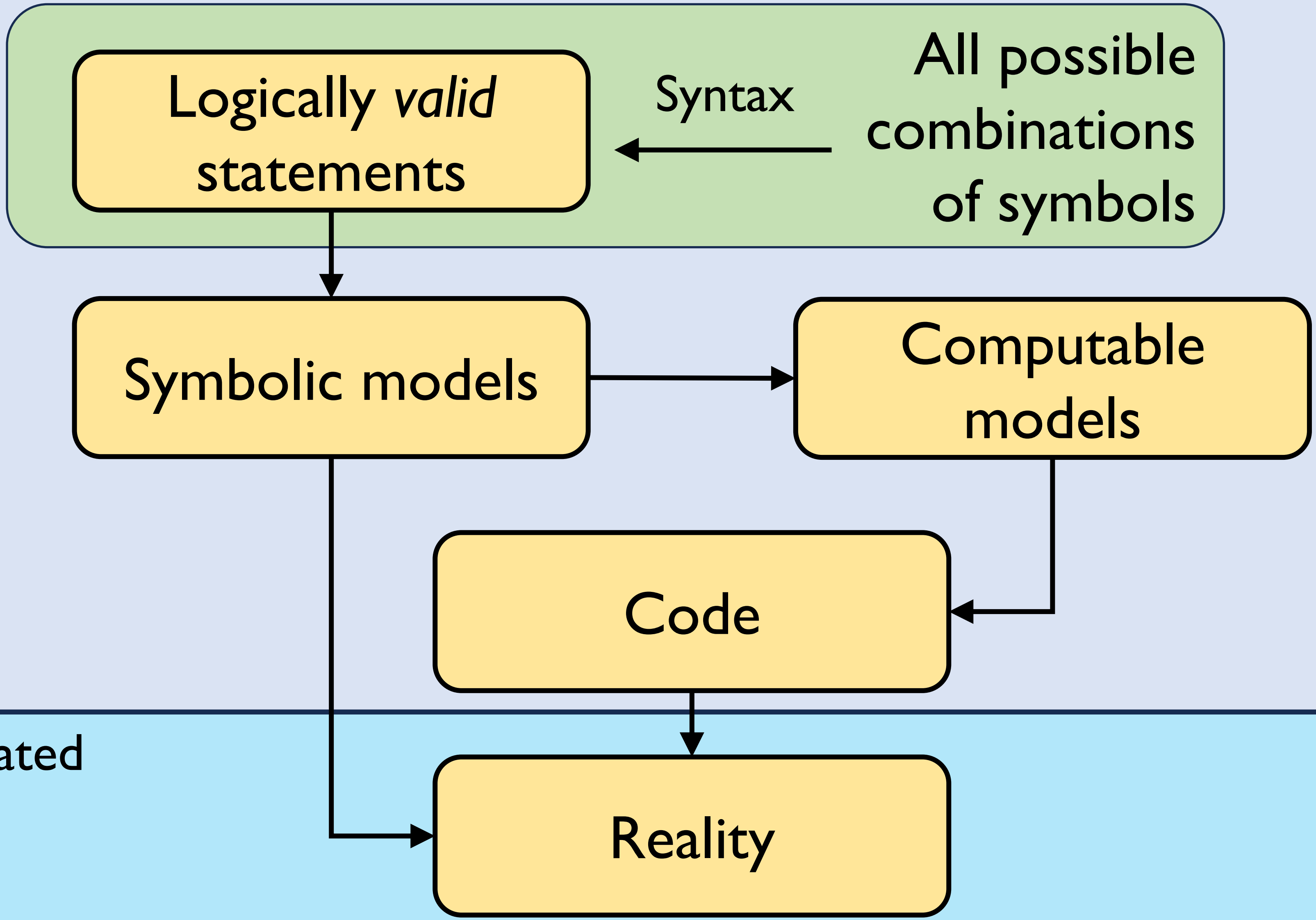
Humans read the theory and write the code as best as they can

Then use various automated and manual means to compare to experiment

Slide from Lecture 3

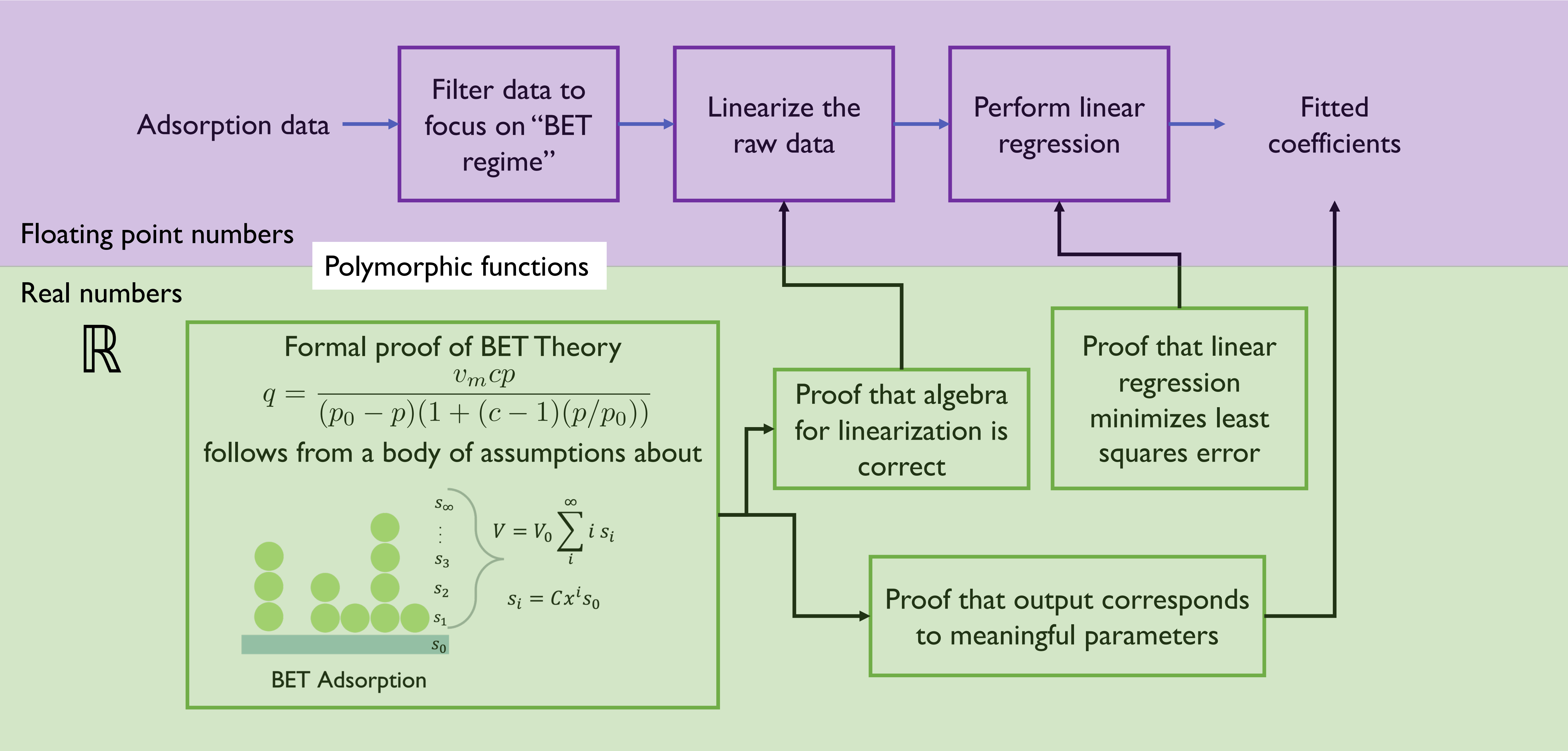
Syntax and semantics in scientific computing

Can we represent all of this in Lean, and validate the construction of the math, scientific models, and software, in one system?



Then use various automated and manual means to compare to experiment

Polymorphic functions to bridge floats and reals



Slide from Lecture 5

Functions: Programming vs. Math

Programming perspective

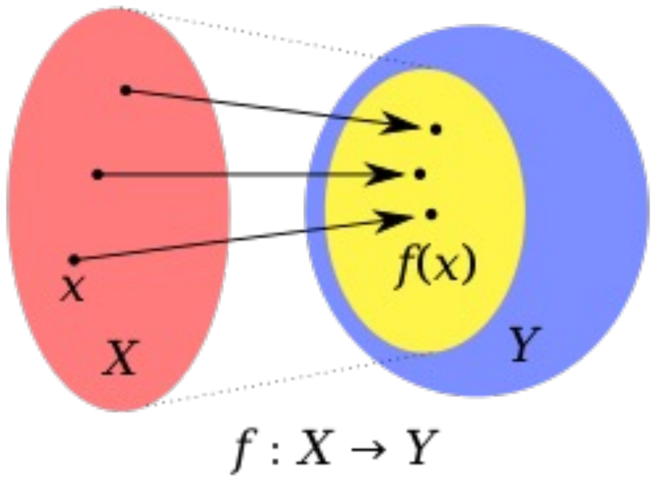
A function takes arguments, performs calculations, and produces an output

Examples in Python

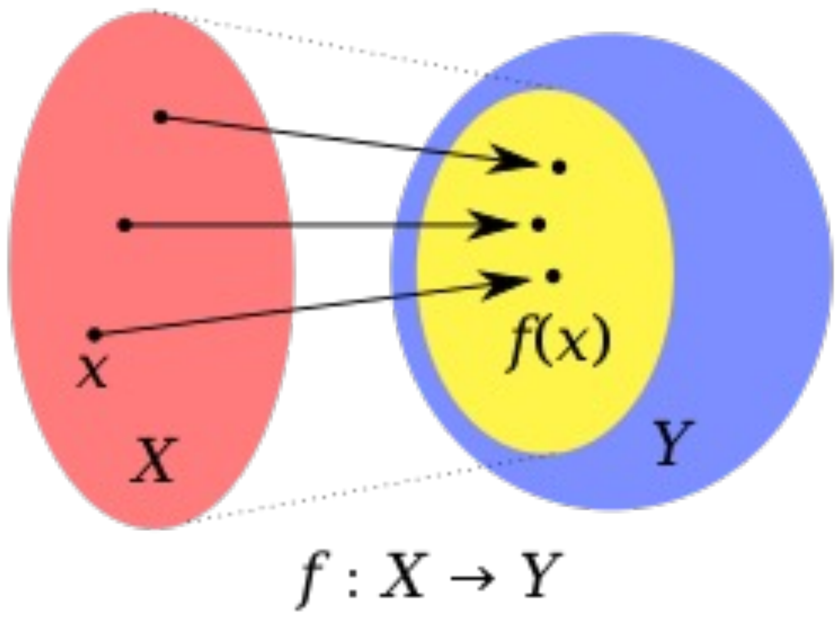
```
def squared(x):
    y = x*x
    return y
```

Math perspective

A function maps values from a domain to a co-domain



Functions: Programming vs. Math



Domain

Co-domain

Image

```
def squareroot(x):
    y = x**(1/2)
    return y
```

$$f(x) = \sqrt{x}$$

Not always a function!
With type $\mathbb{Z} \to \mathbb{Z}$ or $\mathbb{R} \to \mathbb{R}$, there is no mapping from the x < 0 part of the domain

With type $\mathbb{N} \to \mathbb{R}$ or $\mathbb{R} \to \mathbb{C}$, it *is* a function; every part of the domain maps to a value in the co-domain

Slide from Lecture 5

Glossary

- Equation
  - Proposition about equality statement
- Formula
  - Proposition about expressions, includes equalities, inequalities, as well as logical operators
- Expression
  - Like the "right hand side" of an equation
  - Type depends on the types and operations of things inside
- Function (aka pure function)
  - An expression that maps from domain to co-domain
- Partial function
  - An expression that maps from *part* of domain to co-domain

Slide from Lecture 6

Functions in Lean

- Further discussion in Lecture 7
- No parentheses needed – just a space will do
  - f(x) is written as f x
- We can *prove* things about pure functions; it's much harder with partial functions
- Lean requires you to label "noncomputable" functions
  - Noncomputable means "incapable of being computed by any algorithm in a finite amount of time"
  - Real.pi is noncomputable

Slide from Lecture 1

A guide to number systems

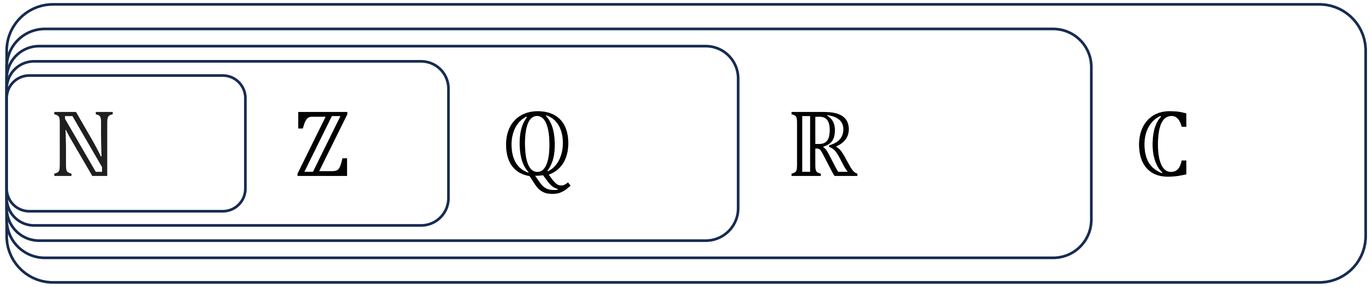
$\mathbb{N}$ - Natural numbers (0, 1, 2, 3, 4, …)

$\mathbb{Z}$ - Integers (… -3, -2, -1, 0, 1, 2, …)

$\mathbb{Q}$ - Rational numbers (1/2, 3/4, 5/9, etc.)

$\mathbb{R}$ - Real numbers (-1, 3.6, π, √2)

$\mathbb{C}$ - Complex numbers (-1, 5 + 2i, √2 + 5i, etc.)



Slide from Lecture 7

Programming Paradigms

Imperative

- Emphasizes *how* to solve
- **State and Mutation**: Variables can be changed after they are set
- **Procedural Style**: Follows a sequence of steps to achieve a result
- **Control Flow**: Uses loops, conditionals, and other control structures
- **Side Effects**: Functions or methods can modify global state or have other side effects
- **Examples**: Python, Java, most languages

Functional

- Emphasizes *what* to solve
- **Immutability**: Variables, once assigned, cannot be changed
- **Declarative Style**: Focuses on defining and declaring what things are
- **Functions Priorit**: Functions can be passed as arguments, returned from other functions, and assigned to variables
- **Pure Functions**: No side effects, given the same input, always produces the same output
- **Examples**: Haskell, Lean 4!

It's possible to write functional-style code in languages like Python

Lean 4 is *purely functional*; it doesn't let you use imperative techniques

Slide from Lecture 7

Why is mutability so popular?

Efficiency

0.61	0.13	0.03
0.27	0.68	0.22
0.22	0.83	0.98
0.15	0.99	0.14
0.24	0.38	0.62
0.46	0.92	0.88
0.41	0.28	0.69
0.58	0.29	0.36
0.68	0.89	0.02
0.89	0.15	0.94

Multiply one element by 2 →

0.61	0.13	0.03
0.27	0.68	0.22
0.22	0.83	0.98
0.15	0.99	0.14
0.24	0.76	0.62
0.46	0.92	0.88
0.41	0.28	0.69
0.58	0.29	0.36
0.68	0.89	0.02
0.89	0.15	0.94

If this matrix is immutable, you need to re-copy the rest of the matrix!

In this case, 2x the memory and 30x the computational cost

Functional programming languages use various tricks to manage cost

Lean 4 introduced the "functional but in-place" paradigm

(see de Moura and Ullrich, CADE 2021 for more details)

Slide from Lecture 7

Recursive functions

- Functions can call other functions
- A function is recursive when *it calls itself*
- Python example: factorial function, n!

Imperative style

```
def factorial_loop(n):
    result = 1
    for i in range(1,n+1):
        result = result*i
    return result
```

Functional style

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

Slide from Lecture 7

Factorial function – recursive

Functional style

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

```
factorial(5)

factorial(5)
5*factorial(5-1)
5*factorial(4)
5*4*factorial(3)
5*4*3*factorial(2)
5*4*3*2*factorial(1)
5*4*3*2*1*factorial(0)
5*4*3*2*1*1

return 120
```

Notice how the "stack" of calculations keeps increasing.
At scale, this creates memory issues.

This means this is not "tail recursive."

Factorial function – tail-recursive

Functional style

```
def factorial_tail(n, acc=1):
    if n == 0:
        return acc
    else:
        return factorial_tail(n-1, n*acc)
```

This tail-recursive function manages the "stack" so it doesn't blow up.

Almost always, tail-recursive functions perform better

```
factorial(5)

factorial(5,1)
factorial(4,5*1)
factorial(4,5)
factorial(3,5*4)
factorial(3,20)
factorial(2,20*3)
factorial(2,60)
factorial(1,60*2)
factorial(1,120)
factorial(0,120)

return 120
```

Slide from Lecture 7

The halting problem

- Let's consider recursive functions
- Does factorial(5) halt?
- How about factorial(20)?
- factorial(1523482)?
- What about factorial(-3)?
- factorial(-60)?

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

You don't need to finish running the program every time

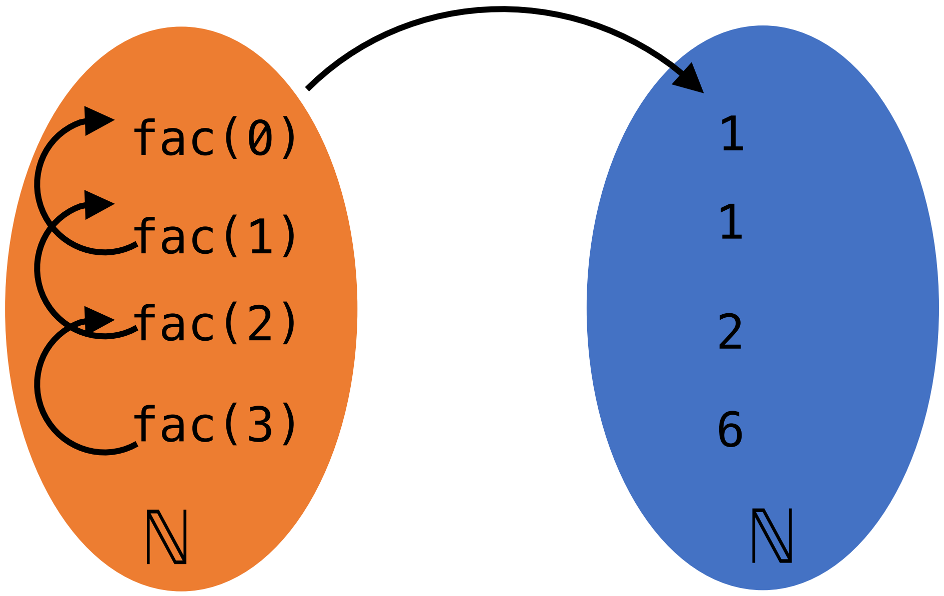
You're using logic to figure this out!

Slide from Lecture 7

Recursion in Lean

This function works

```
def factorial : ℕ → ℕ
  | 0 => 1
  | n + 1 => (n + 1) ∗ factorial n
```



This function is broken

```
def not_factorial : ℕ → ℕ
  | 0 => 1
  | n + 1 => (n + 1) ∗ not_factorial (n+1)
```

Check out the error message on not_factorial:

```
fail to show termination for not_factorial
with errors
structural recursion cannot be used:
```

In factorial, Lean automatically proves termination via structural recursion, so this function is okay.

“Do” notation in Lean

- Lean *can* express imperative-style programs using “do” notation
- Helpful if you just want to write programs, but this makes proof-writing much more difficult

```
def factorial_do (n : Nat) : Nat := Id.run do
  let mut result := 1
  for i in [1:n+1] do
    result := result * i
  return result
```

Polymorphic functions

- *Polymorphism* is when a single symbol represents different types
- A *polymorphic function* takes variables that can be more than one type
- Python uses polymorphism (most languages do), so a relatively short list of familiar symbols can address diverse tasks

```
def plus(a,b):
    return a + b
```

```
plus(1,2)
3
```

```
plus(1.0,2.0)
3.0
```

```
plus('1','2’)
'12'
```

```
plus([1],[2])
[1, 2]
```

```
plus(1.0,2)
3.0
```

Polymorphism in Python is [ad hoc] – under the hood, these are compiled as distinct functions

Polymorphism in Lean

- In functional programming languages, polymorphism is made possible using generic types, which get inhabited by specific types based on context

- For example, let's revisit the structure Point from last time
- We can define a similar structure PPoint that's polymorphic (from FPIL 1.6)

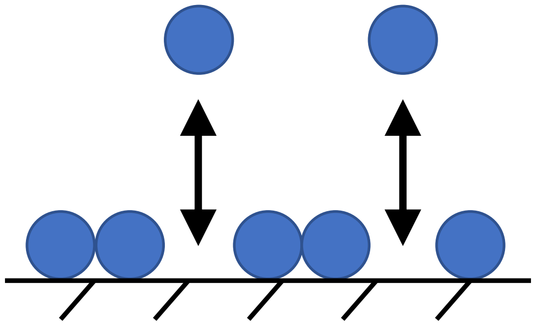
```
structure Point where
  x : Float
  y : Float
deriving Repr
```

```
structure PPoint (α : Type) where
  x : α
  y : α
deriving Repr
```

Polymorphic functions

- Three case studies
  - identity
  - plusOne
  - Langmuir

Derivations in science are math proofs



Langmuir Adsorption

Langmuir, *JACS*, 1918

Proposition

5 premises — imply → **conjecture**

Site balance:	$S_0 = S + S_\mathrm{a}$
Adsorption rate model:	$r_\mathrm{ads} = k_\mathrm{ads} \cdot p \cdot S$
Desorption rate model:	$r_\mathrm{des} = k_\mathrm{des} \cdot S_\mathrm{a}$
Equilibrium assumption:	$r_\mathrm{ads} = r_\mathrm{des}$
Mass balance	$q = S_\mathrm{a}$

$$q = \frac{S_0 K_{eq} p}{1 + K_{eq} p}$$

Theorem

Proposition is TRUE

Proof

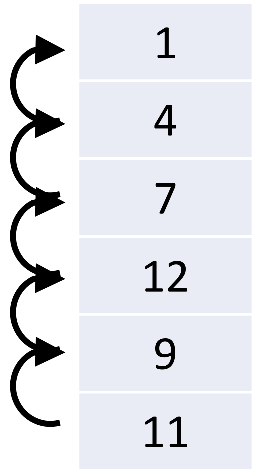
Derivation using algebraic manipulations (substitution, cancelling terms, etc.)

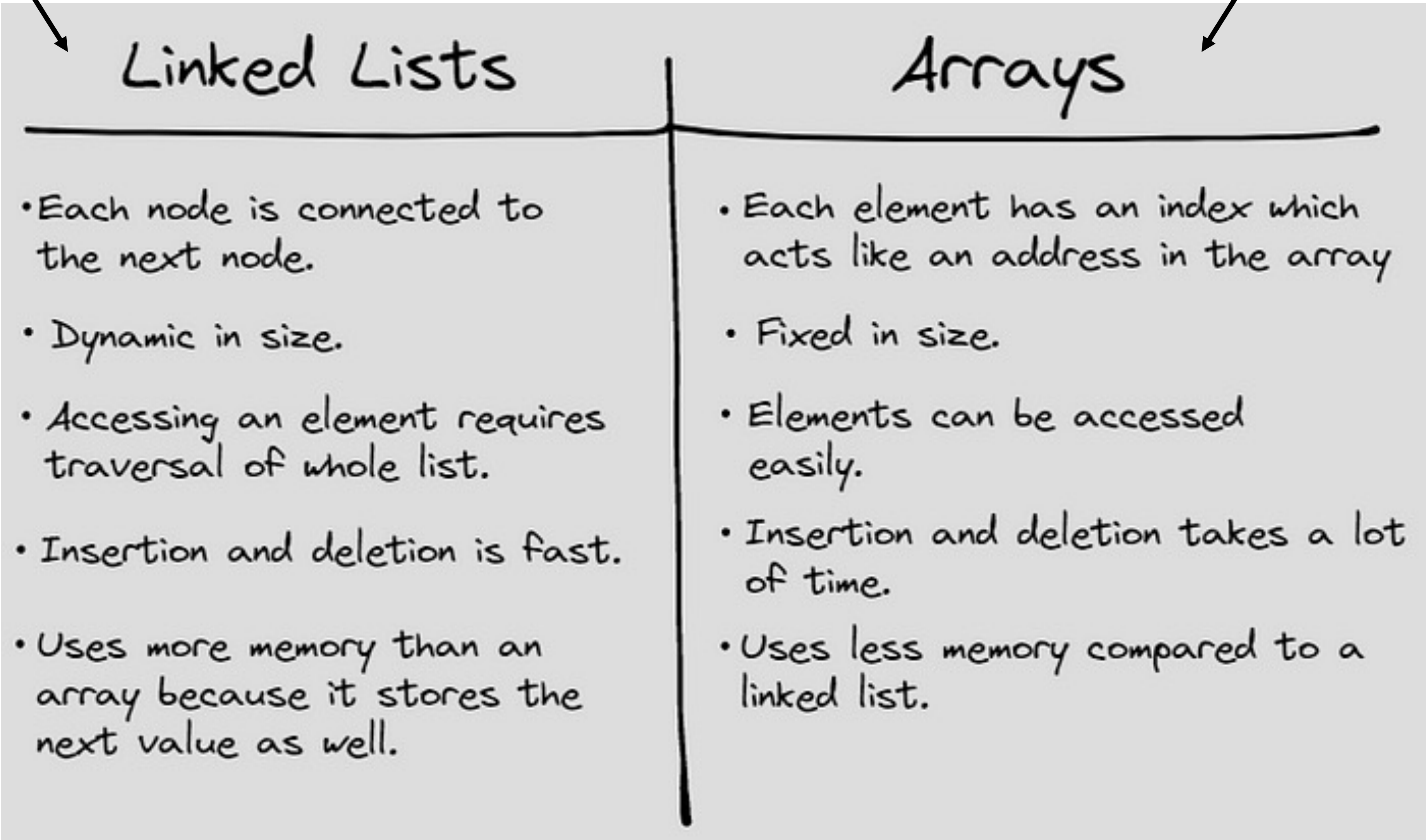
✓ ____
✓ ____
✓ ____
✓ ____

Lists vs Arrays

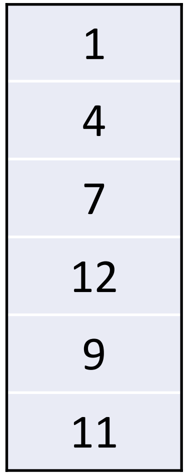
A "list" in Lean is a linked list

A "list" in Python is an array!





Linked Lists	Arrays
• Each node is connected to the next node.	• Each element has an index which acts like an address in the array
• Dynamic in size.	• Fixed in size.
• Accessing an element requires traversal of whole list.	• Elements can be accessed easily.
• Insertion and deletion is fast.	• Insertion and deletion takes a lot of time.
• Uses more memory than an array because it stores the next value as well.	• Uses less memory compared to a linked list.



https://medium.com/@bilal_k/wtf-is-linked-list-5d58b8a3bfe7

Lists in Lean

- FPIL Ch 3
- Lists in Lean are linked lists
- When you declare them, you need to specify the type of the data included, or specify a generic type and use polymorphism

```
def primesUnder10 : List Nat := [2, 3, 5, 7]

def periodicTable : List String :=
  ["H", "He", "Li", "Be", "B", "C", "N", "O", "F", "Ne”]
```

- Summing elements in a list requires by polymorphism and recursion

```
def sum_list : List Nat → Nat
| [] => 0
| (x :: xs) => x + sum_list xs
```

Lists and Arrays in Lean