Chemical Biochemical and Environmental Engineering

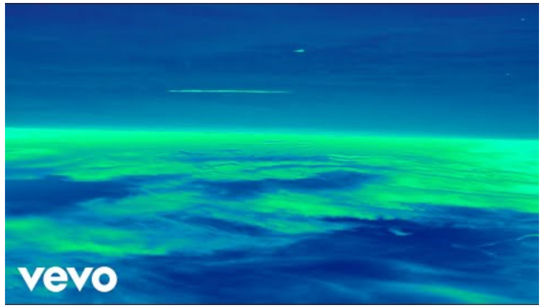
LEAN THEOREM PROVER

Lean for Scientists and Engineers

Tyler R. Josephson

AI & Theory-Oriented Molecular Science (ATOMS) Lab

University of Maryland, Baltimore County



Dream Dream Dream
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

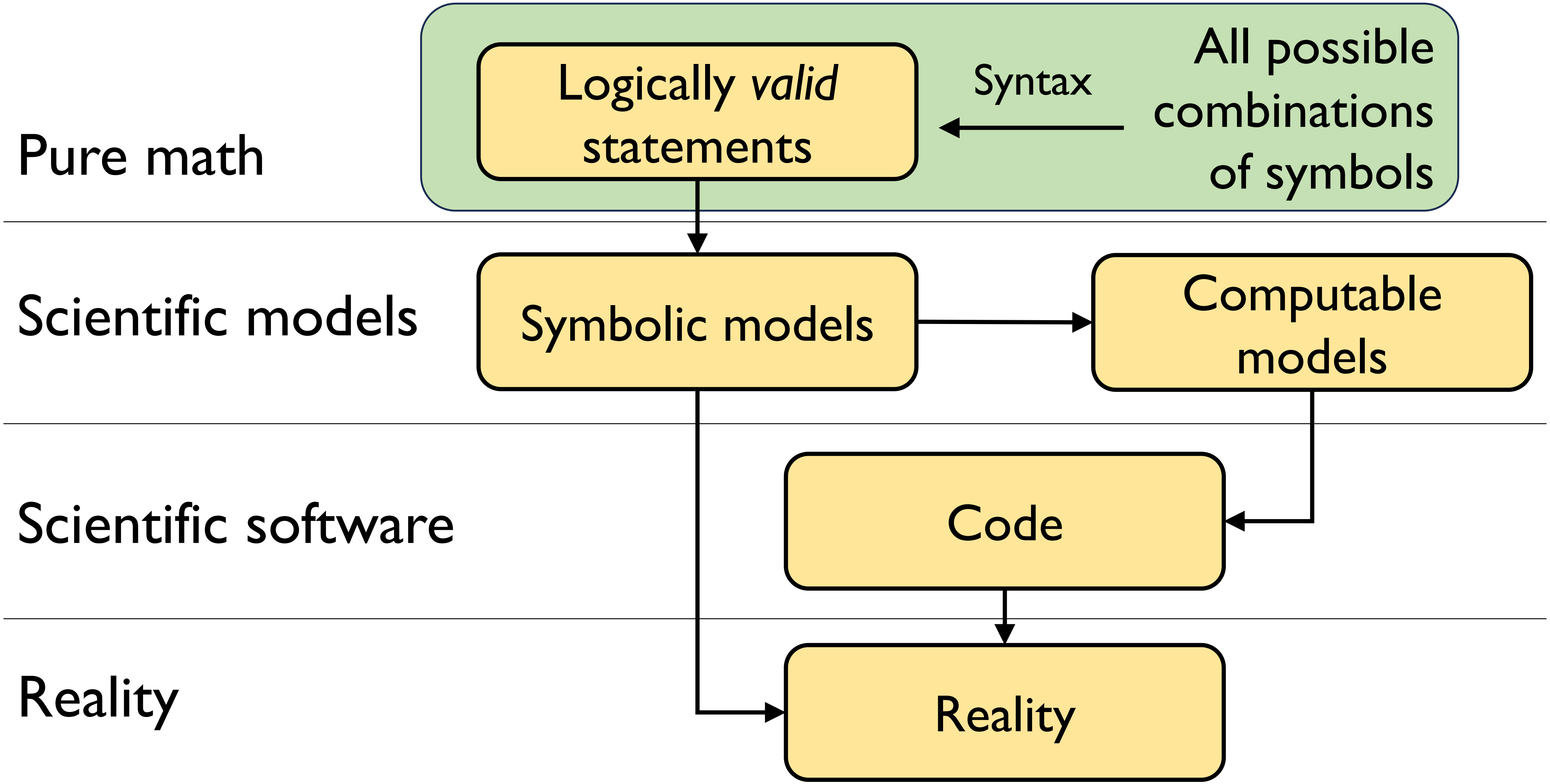
Big thanks to David Christiansen and Functional Programming in Lean!

Schedule for today

- Recap Lecture 8
- Lists
  - Defining lists
  - Accessing and slicing elements
  - Applying functions
  - Recursion over lists
  - Filtering using if ... then
  - Folding
- Vectors
- Strings
- Input/output
  - An analogy
  - Hello, world!
- Compiling Lean to C
- Example: CSV Parser

Slide from Lecture 3

Syntax and semantics in scientific computing



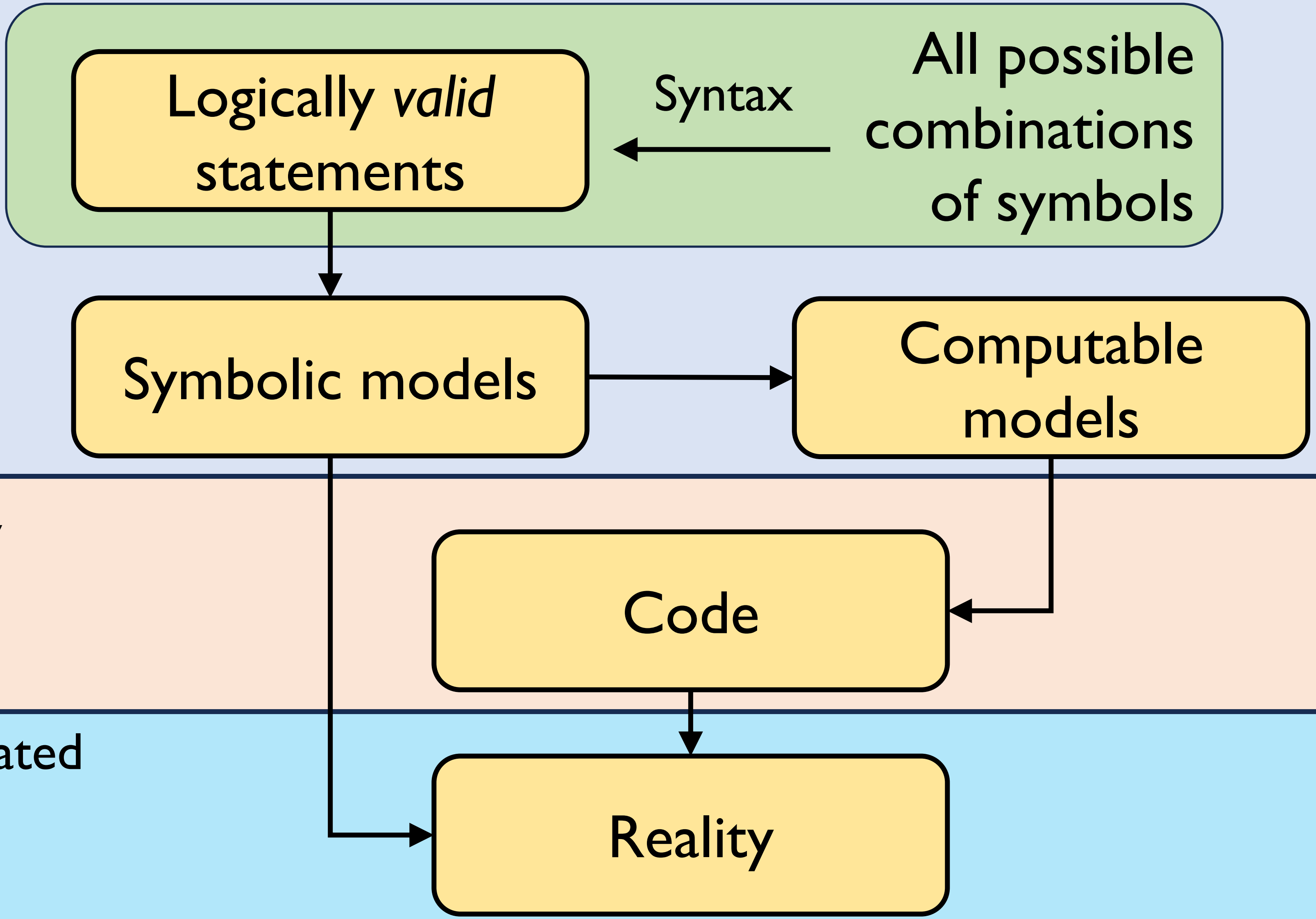
Slide from Lecture 3

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

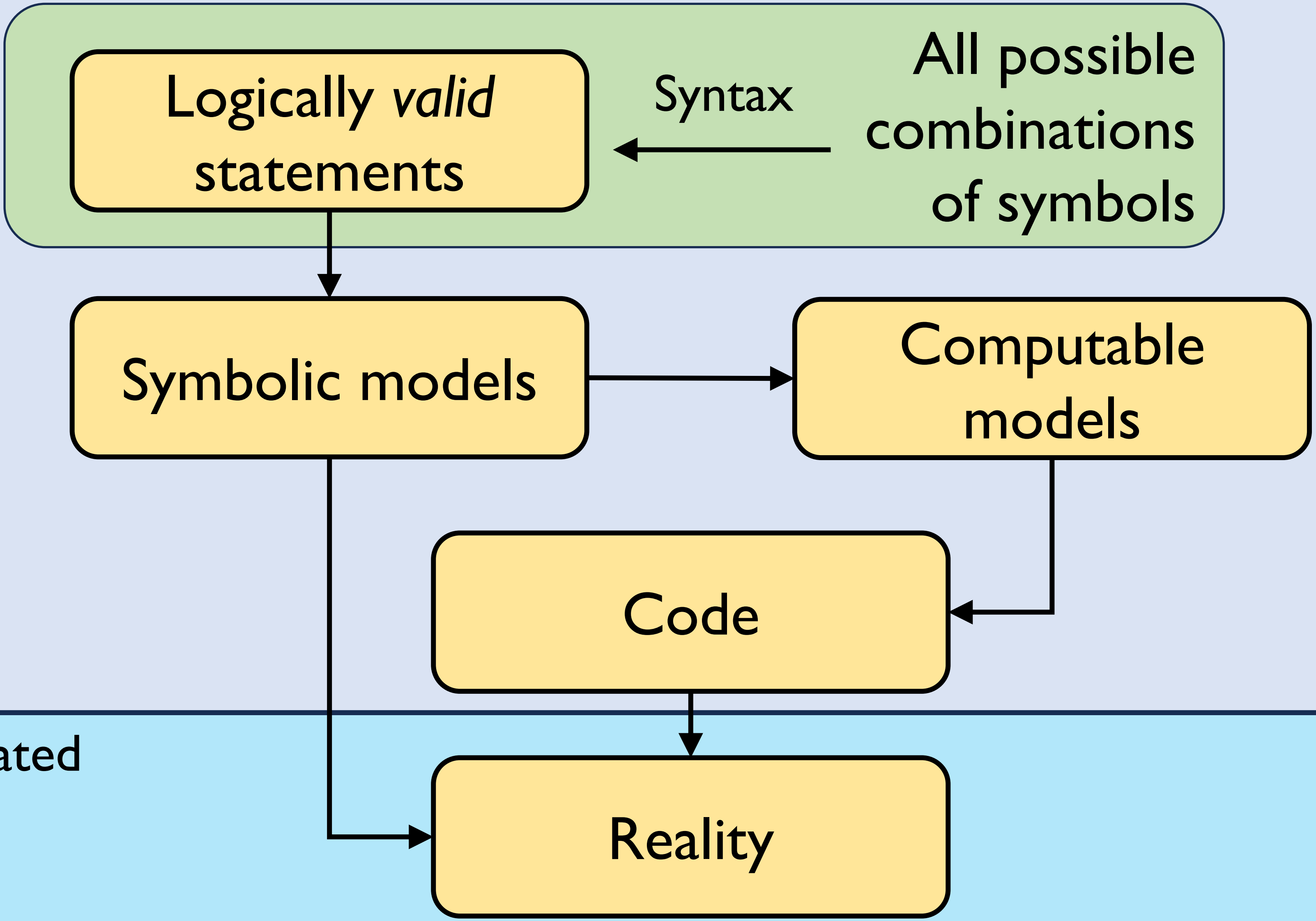
Then use various automated and manual means to compare to experiment



Slide from Lecture 3

Syntax and semantics in scientific computing

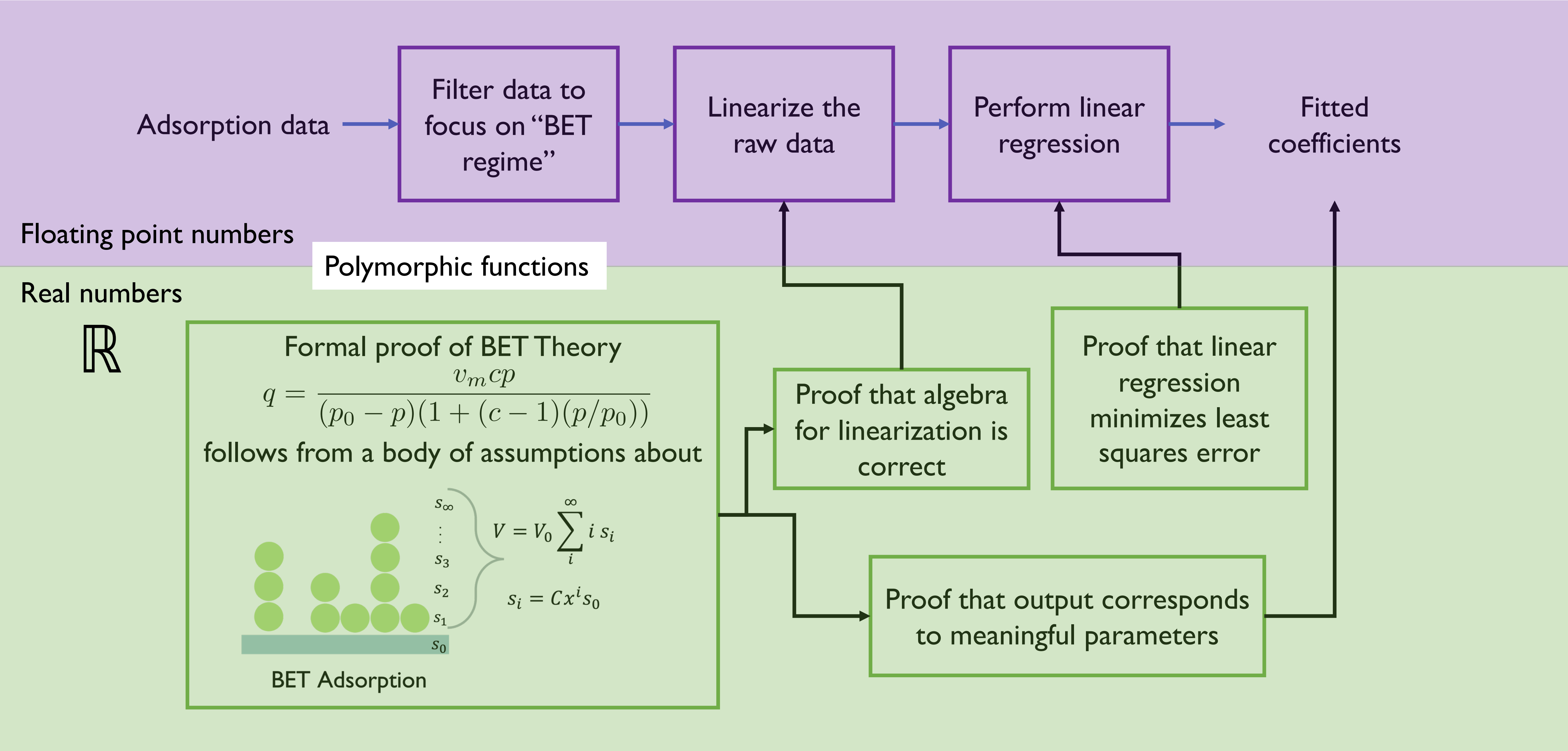
Can we represent all of this in Lean, and validate the construction of the math, scientific models, and software, in one system?



Then use various automated and manual means to compare to experiment

Slide from Lecture 1

Polymorphic functions to bridge floats and reals



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 Prioritized**: 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."

Slide from Lecture 7

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)
```

```
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
```

This tail-recursive function manages the “stack” so it doesn’t blow up.

Almost always, tail-recursive functions perform better

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)
```

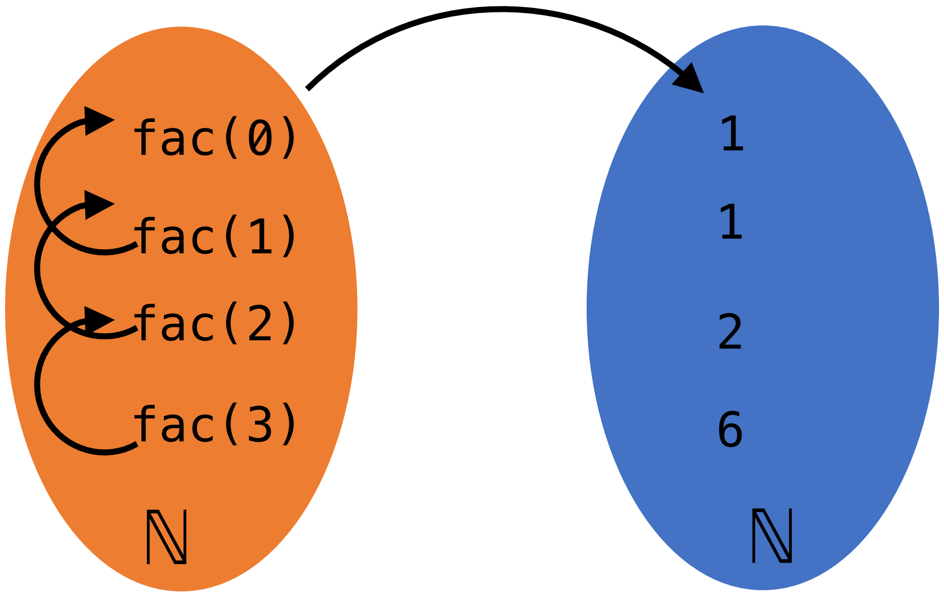
You don't need to finish running the program every time
You're using logic to figure this out!

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:

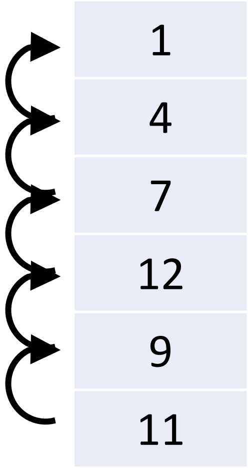
```
fail to show termination for not_factorial
with errors
structural recursion cannot be used:
```

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

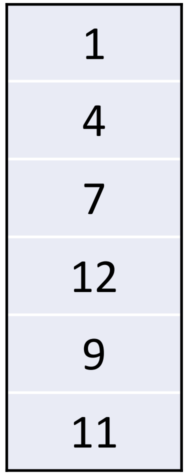
Lists vs Arrays

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

Slide from Lecture 8

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 recursion

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

Lists

Vectors

- Lean is a dependently-typed programming language
- Types can depend on values
- The type "Vector" is a List with the list length as a value

```
def Vector (α : Type ) (n : ℕ) :=
  { l : List α // l.length = n }
```

Strings

- Strings defined using double quotations, e.g. `"hello"`
- Single quotations are used for characters, e.g. `'c'`

- Go to StringExamples.lean

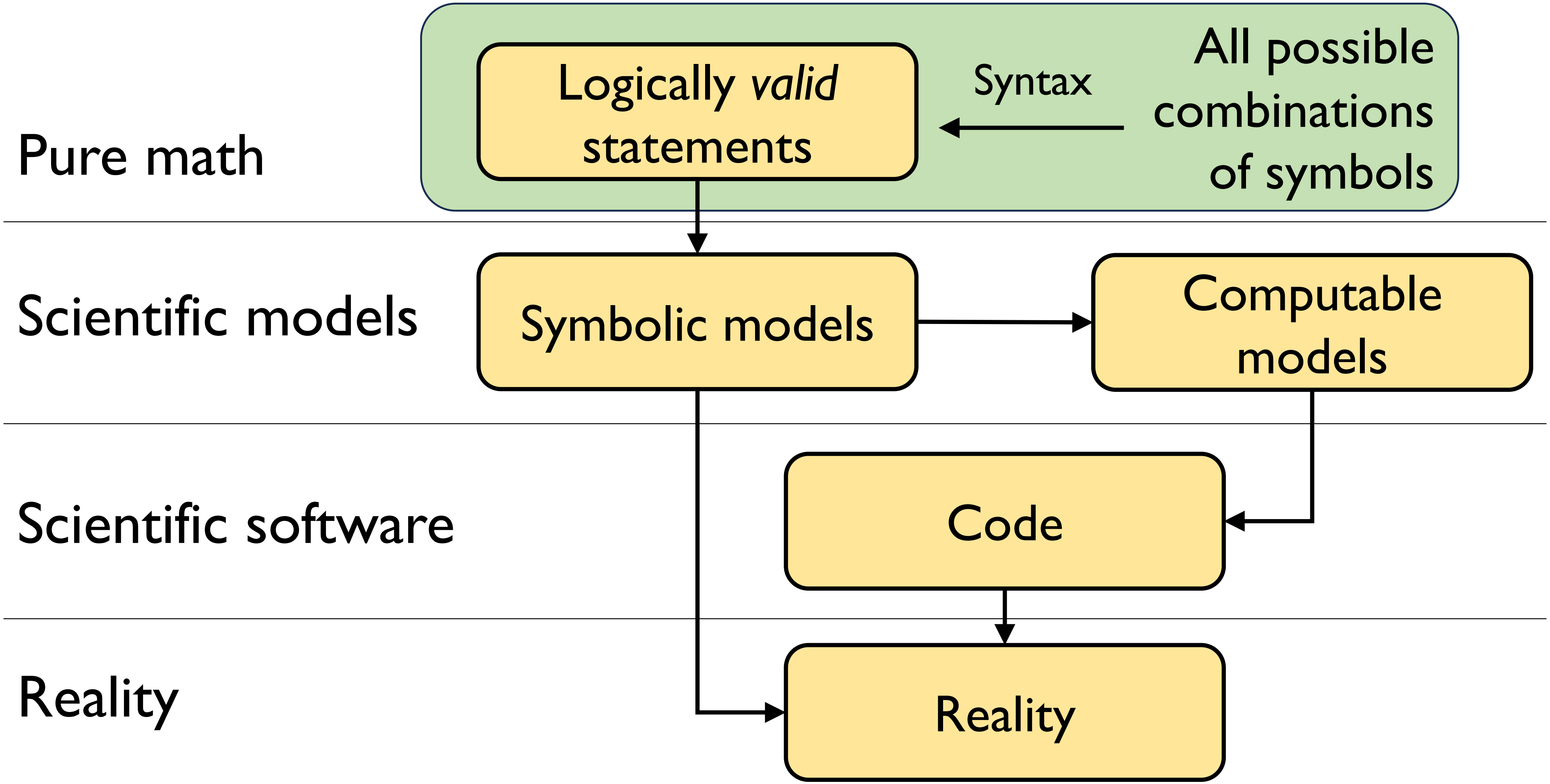
Input / Output: An Analogy

Kitchen (back of house)	Waiter	Dining room (front of house)

Pure functions Mathlib Verified logical syntax	IO Monad	Messy, unpredictable real world

Slide from Lecture 3

Syntax and semantics in scientific computing



Basic Output

```
def main : IO Unit := IO.println "Hello, world!"
```

Save file as Hello.lean

Then, from the Terminal, run:

```
lean --run Hello.lean
```

Input + Output

```
def main : IO Unit := do
  let stdin ← IO.getStdin
```

let introduces a variable that will be assigned as such throughout the do block

← Using an arrow means that the value of the expression is an IO action that should be executed, with the result of the action saved in the local variable.

1. Evaluate the right-hand side
2. Learn that it's an IO action: IO.getStdin
3. Execute this IO action to create a value: stdin
4. stdin has type IO.FS.Stream

```
  let input ← stdin.getLine
```

1. Evaluate the right-hand side
2. Learn that it's an IO action: getLine
3. Execute this IO action to create a value: input
4. input has type String

`:=` is used here instead of ← because the right side is pure functions, not IO

```
  IO.println "How would you like to be addressed?"
  let name := input.dropRightWhile Char.isWhitespace
  IO.println s!"Hello, {name}!"
```

Input + Calculation + Output

- Let's get a number from standard input and calculate its factorial!
- But wait, what happens if standard input doesn't provide a number?
- We need a way to manage error handling

- Option Monad

Opportunity for further study: Monads

- IO is a "monad"
- My favorite resource introducing Monads: https://www.youtube.com/watch?v=Q0aVbqim5pE
- A helpful analogy from Scott Wlaschin (code in F#): https://vimeo.com/113588389 (timestamp 38:48)

Compiling Lean to C

- Lean 4 is designed to be able to be compiled to C

```
lean --c=cfile.c leanfile.lean
```

Example: CSV Parser

- CSV: comma-separated values
- Chris Lovett from Microsoft wrote CSV parser

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