# CSCI 5535 Fundamentals of Programming Languages

Aka How not to fear proofs!

## Lec 1: Introduction

CUPLV

CU Programming Languages
& Verification

# About me

- Assistant Professor, Dept. of Computer Science
- Research: Programming Languages and Formal Methods. Applications in Distributed Systems, Cryptography, and Security.
- In my free time:
  - I read: History, Biographies, Pop-science, Detective Fiction, …
  - I hike: Davidson Mesa & Coal Creek Trail
  - I play: silly games with my 2-y-o daughter.

Gowtham Kaki

Pronounced

*g-OW-thum*

close enough!

# About CSCI 5535 / ECEN 5533

1. Learn to make *precise* statements and *prove* them.
2. Learn to use this skill to study *mathematical foundations* of computer programming.

# Haven’t we always been making precise statements in math?

“There do not exist four positive integers, the last being greater than two, such that the sum of the first two, each raised to the power of the fourth, equals the third raised to that same power.”

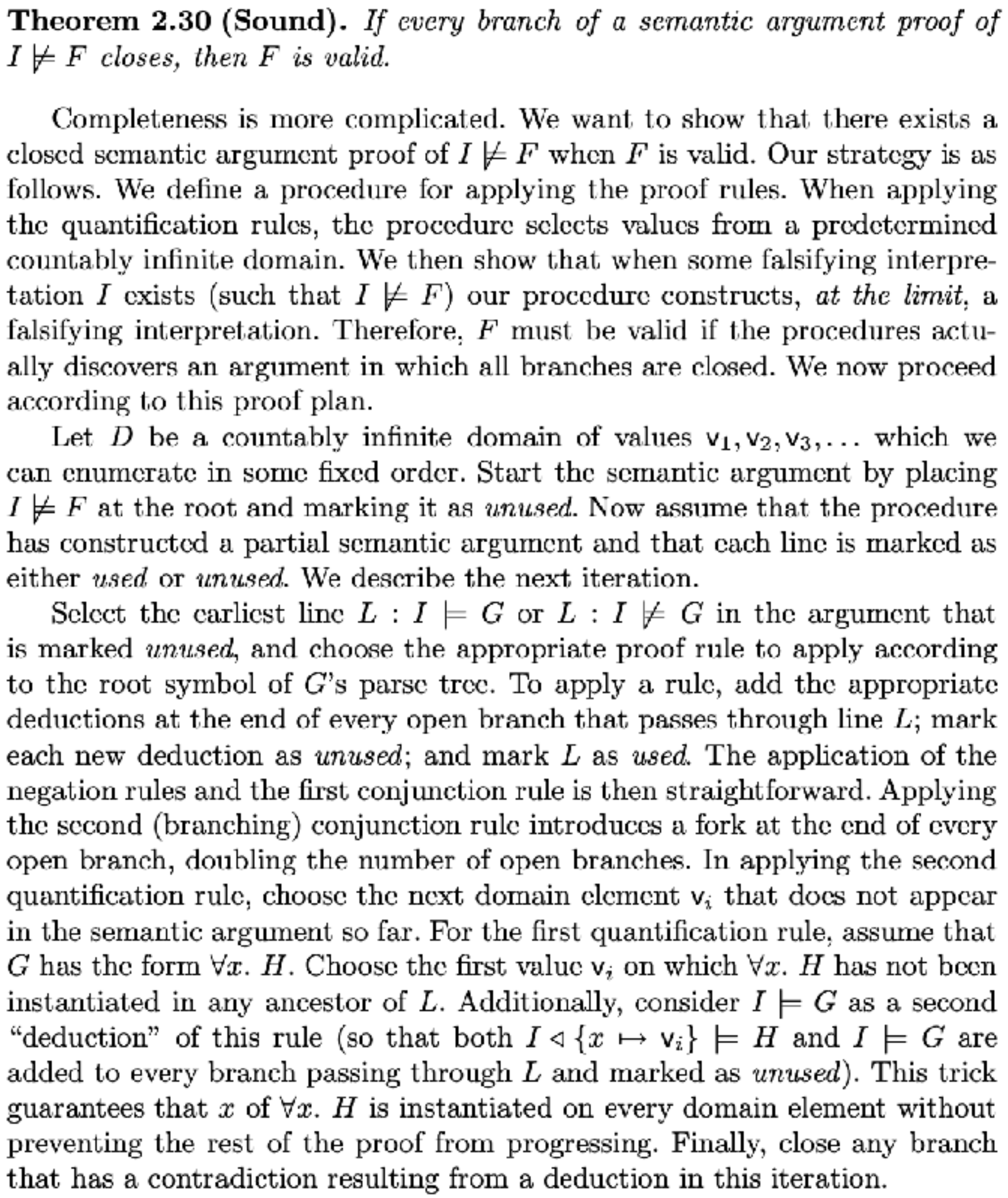
Vs

There do not exist positive integers $x, y, z,$ and $n$, with $n > 2$, such that $x^n + y^n = z^n$

Vs

$$\nexists(x \in \mathbb{N}, y \in \mathbb{N}, z \in \mathbb{N}, n \in \mathbb{N}) \,.\, n > 2 \land x^n + y^n = z^n$$

# Haven’t we always been writing proofs in math?



**Theorem 2.30 (Sound).** *If every branch of a semantic argument proof of $I \not\models F$ closes, then $F$ is valid.*

Completeness is more complicated. We want to show that there exists a closed semantic argument proof of $I \not\models F$ when $F$ is valid. Our strategy is as follows. We define a procedure for applying the proof rules. When applying the quantification rules, the procedure selects values from a predetermined countably infinite domain. We then show that when some falsifying interpretation $I$ exists (such that $I \not\models F$) our procedure constructs, *at the limit*, a falsifying interpretation. Therefore, $F$ must be valid if the procedures actually discovers an argument in which all branches are closed. We now proceed according to this proof plan.

Let $D$ be a countably infinite domain of values $\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \ldots$ which we can enumerate in some fixed order. Start the semantic argument by placing $I \not\models F$ at the root and marking it as *unused*. Now assume that the procedure has constructed a partial semantic argument and that each line is marked as either *used* or *unused*. We describe the next iteration.

Select the earliest line $L : I \models G$ or $L : I \not\models G$ in the argument that is marked *unused*, and choose the appropriate proof rule to apply according to the root symbol of $G$'s parse tree. To apply a rule, add the appropriate deductions at the end of every open branch that passes through line $L$; mark each new deduction as *unused*; and mark $L$ as *used*. The application of the negation rules and the first conjunction rule is then straightforward. Applying the second (branching) conjunction rule introduces a fork at the end of every open branch, doubling the number of open branches. In applying the second quantification rule, choose the next domain element $\mathsf{v}_i$ that does not appear in the semantic argument so far. For the first quantification rule, assume that $G$ has the form $\forall x.\ H$. Choose the first value $\mathsf{v}_j$ on which $\forall x.\ H$ has not been instantiated in any ancestor of $L$. Additionally, consider $I \models G$ as a second “deduction” of this rule (so that both $I \lhd \{x \mapsto \mathsf{v}_i\} \models H$ and $I \models G$ are added to every branch passing through $L$ and marked as *unused*). This trick guarantees that $x$ of $\forall x.\ H$ is instantiated on every domain element without preventing the rest of the proof from progressing. Finally, close any branch that has a contradiction resulting from a deduction in this iteration.

What’s wrong with this?

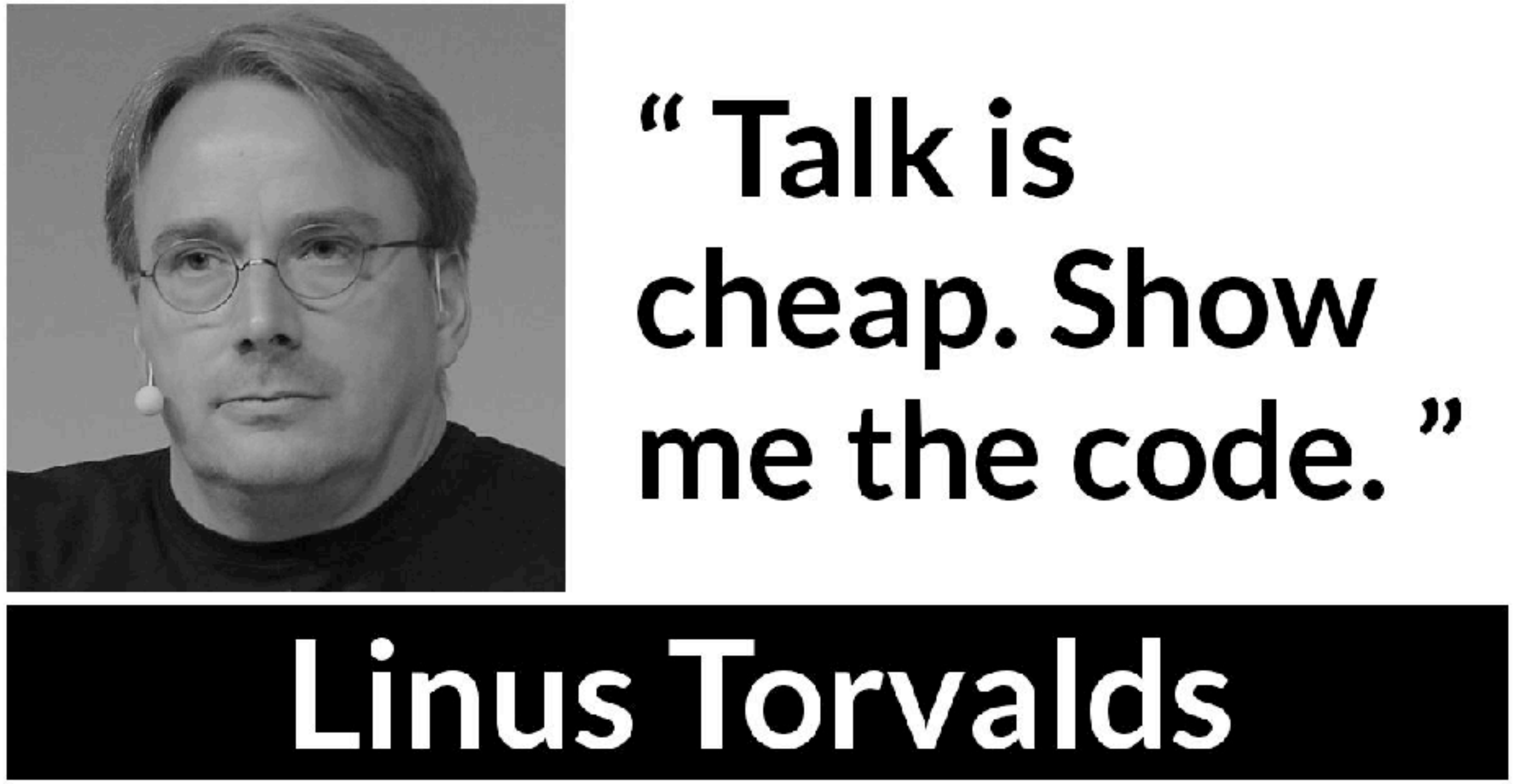
- Informal arguments
- Hard to check
- Incomprehensible
- Error-prone

“Anecdotal evidence suggests that as many as a third of all papers published in mathematical journals contain mistakes—not just minor errors, but incorrect theorems and proofs” — Lamport, How to Write a Proof, 1993.

Proofs ~ Programs

Writing a proof in English ~ Writing a program in English

# What is code for math?



“ Talk is cheap. Show me the code. ”

Linus Torvalds

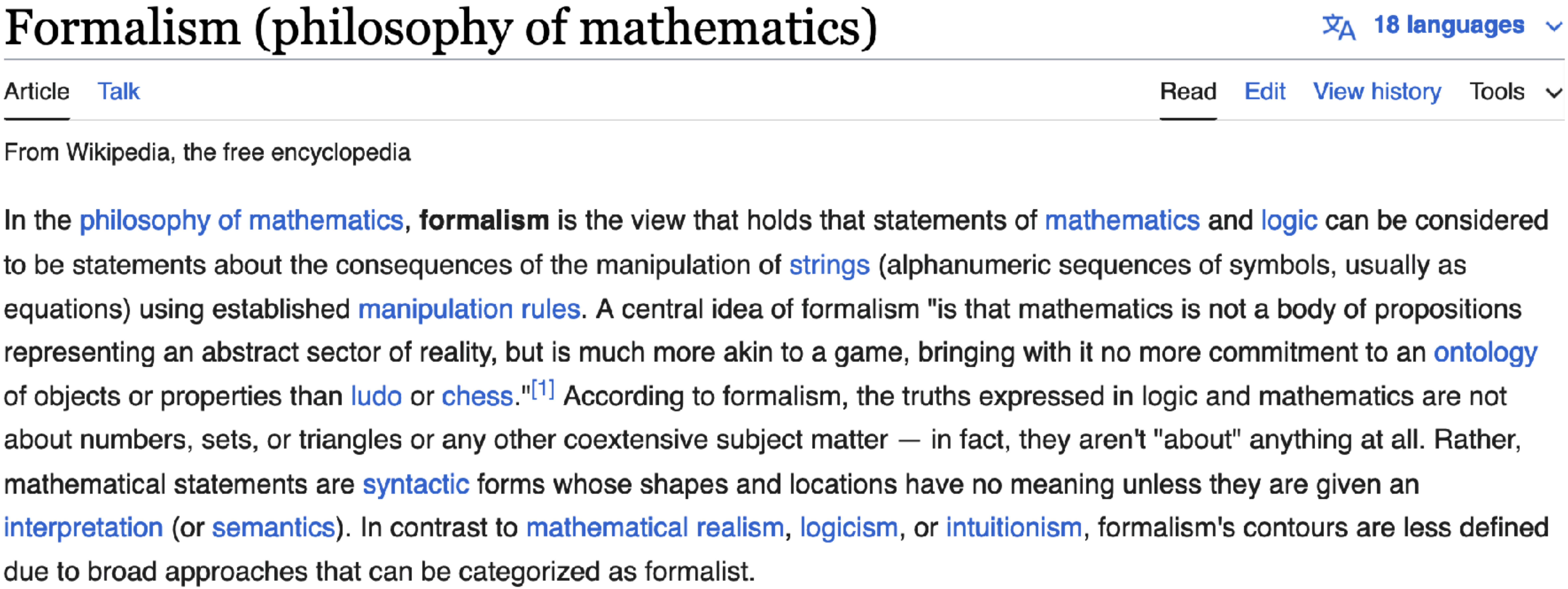
Code is/has:

- Precise semantics
- Executable by a machine
- Understandable by a human
- Composable
- Fails if incorrect
- Means the same everywhere*

Formal Mathematics has all these qualities!

“Formal mathematics is nature's way of letting you know how sloppy your mathematics is.” — Lamport, Specifying Systems, 2002.

# Formal Mathematics

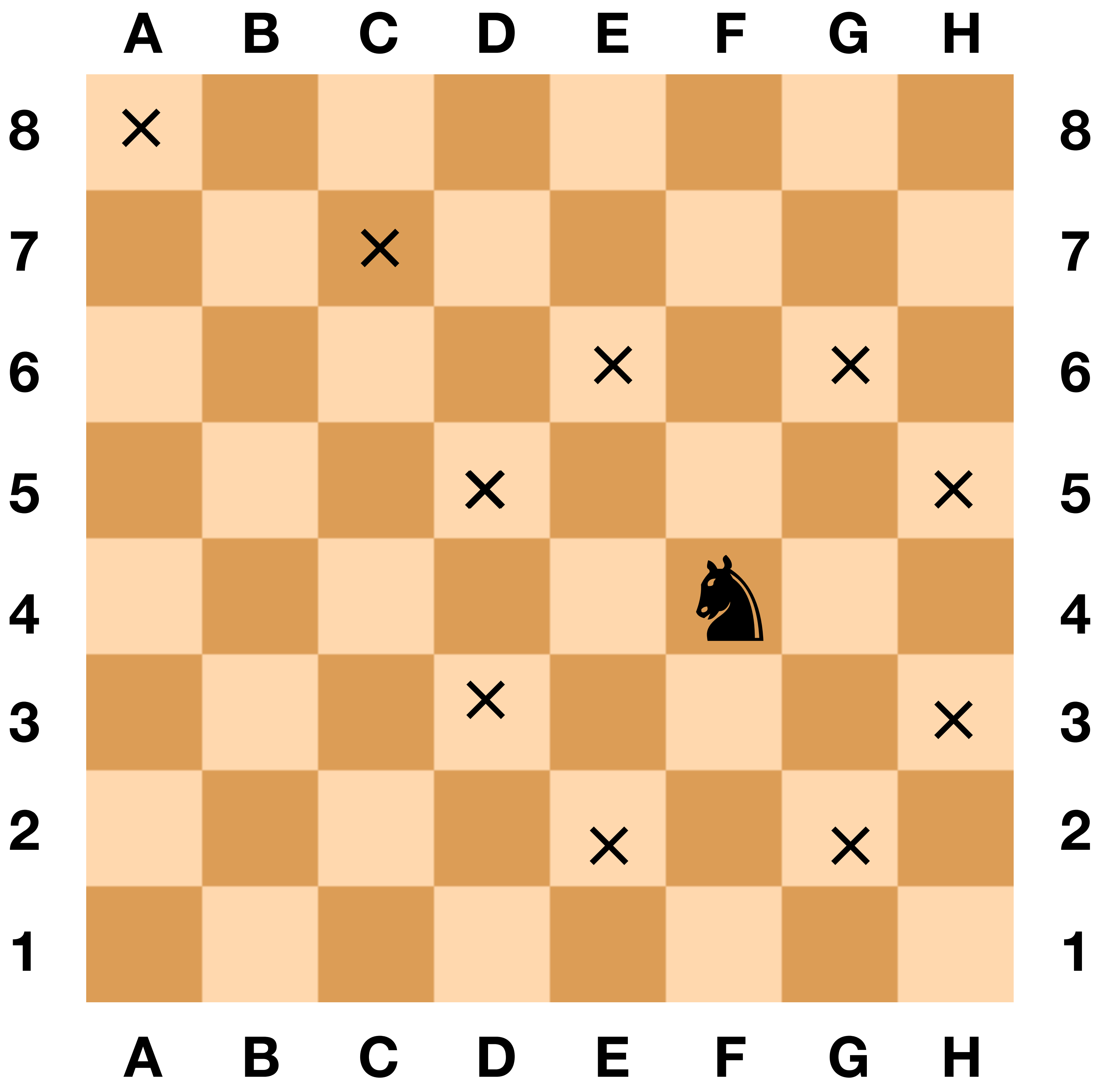


## Formalism (philosophy of mathematics)

From Wikipedia, the free encyclopedia

In the philosophy of mathematics, **formalism** is the view that holds that statements of mathematics and logic can be considered to be statements about the consequences of the manipulation of strings (alphanumeric sequences of symbols, usually as equations) using established manipulation rules. A central idea of formalism "is that mathematics is not a body of propositions representing an abstract sector of reality, but is much more akin to a game, bringing with it no more commitment to an ontology of objects or properties than ludo or chess."[1] According to formalism, the truths expressed in logic and mathematics are not about numbers, sets, or triangles or any other coextensive subject matter — in fact, they aren't "about" anything at all. Rather, mathematical statements are syntactic forms whose shapes and locations have no meaning unless they are given an interpretation (or semantics). In contrast to mathematical realism, logicism, or intuitionism, formalism's contours are less defined due to broad approaches that can be categorized as formalist.

# Formal proof: an analogy



<u>Definition:</u> $CR(i, j)$: Knight CanReach the square $\langle i, j \rangle$

<u>Theorem</u>: $CR(A{,}8) \Rightarrow CR(F{,}4)$

$\Downarrow$ *NamePremise* $P_1$

$\langle P_1 : CR(A{,}8) \rangle \quad \vdash \quad CR(F{,}4)$

$\Downarrow$ *Invert* $CR(F{,}4)$

$\langle P_1 : CR(A{,}8) \rangle \vdash CR(D{,}5) \vee CR(E{,}6) \vee CR(G{,}6) \vee CR(H{,}5)$

$\vee\ CR(H3) \vee CR(G2) \vee CR(E{,}2) \vee CR(D3)$

$\Downarrow$ *PickDisjunct* $CR(D{,}5)$

$\langle P_1 : CR(A{,}8) \rangle \vdash CR(D{,}5)$

$\Downarrow$ *Invert* $CR(D{,}5)$

$\langle P_1 : CR(A{,}8) \rangle \vdash CR(C{,}7) \vee CR(B{,}6) \vee \ldots$

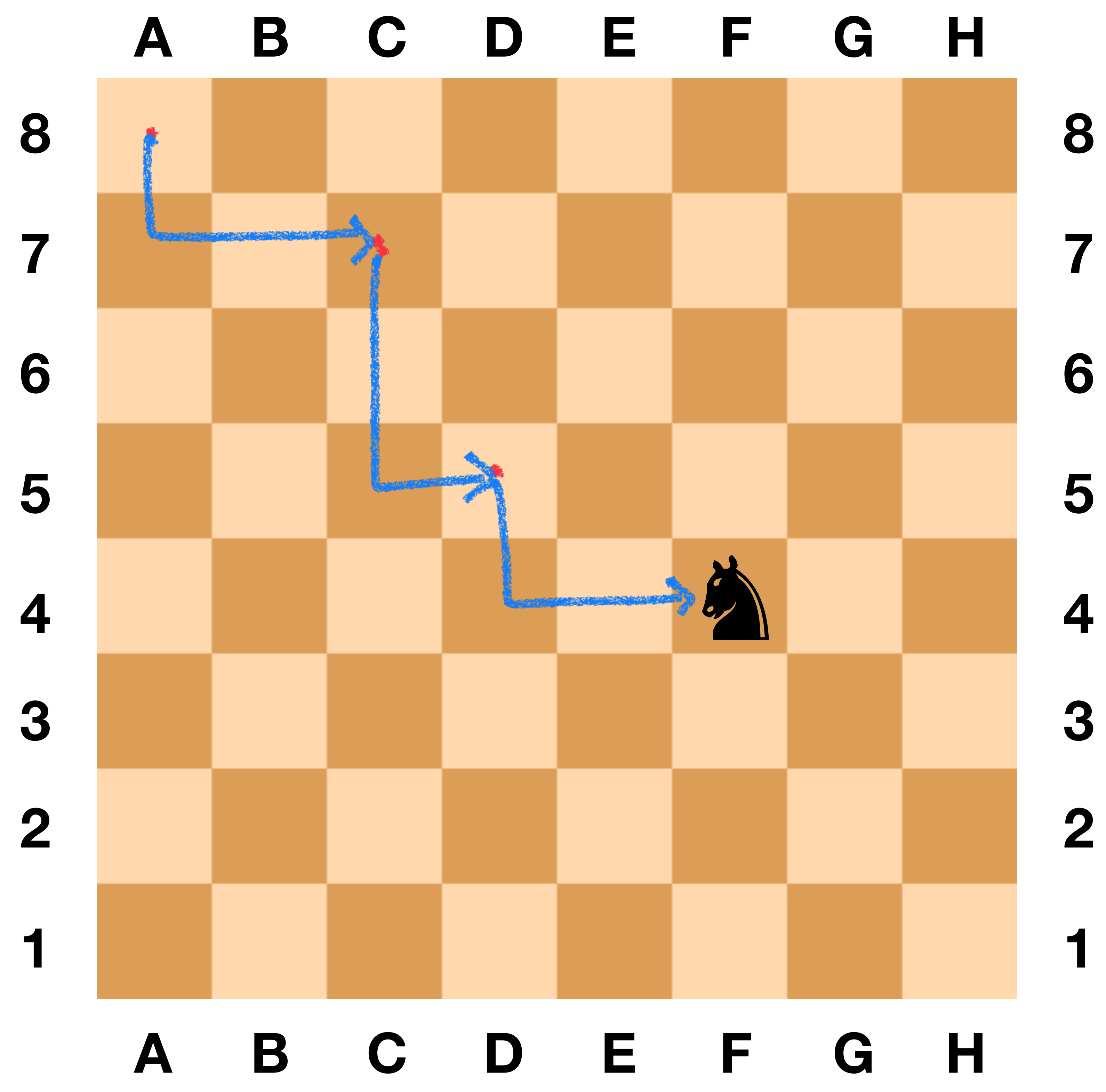
$\Downarrow$ *PickDisjunct* $CR(C{,}7)$

$\langle P_1 : CR(A{,}8) \rangle \vdash CR(C{,}7)$ $\qquad\qquad\qquad\qquad$ *true*

$\Downarrow$ *Invert* $CR(C{,}7)$ $\qquad\qquad\qquad\qquad$ $\Uparrow$ *ApplyPremise* $P_1$

$\langle P_1 : CR(A{,}8) \rangle \vdash CR(A{,}8) \vee CR(A{,}6) \vee CR(E.8) \vee CR(E{,}6)$ $\ \Rightarrow$ *PickDisjunct* $CR(A{,}8)$ $\ \langle P_1 : CR(A{,}8) \rangle \vdash CR(A{,}8)$

# Formal proof: an analogy



Definition: $CR(i, j)$: Knight CanReach the square $\langle i, j \rangle$

Theorem: $CR(A,8) \Rightarrow CR(F,4)$

*NamePremise* $P_1$
*Invert* $CR(F,4)$
*PickDisjunct* $CR(D,5)$
*Invert* $CR(D,5)$
*PickDisjunct* $CR(C,7)$
*Invert* $CR(C,7)$
*PickDisjunct* $CR(A,8)$
*ApplyPremise* $P_1$

A machine-checkable *proof script*!

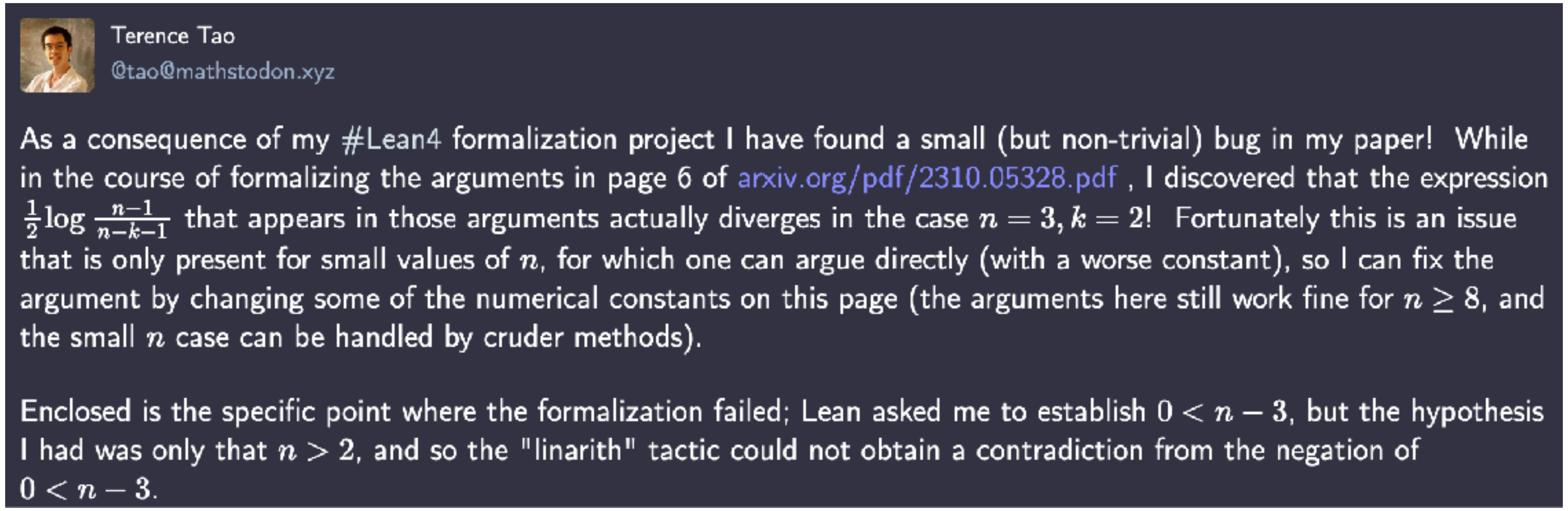
We will apply the same idea to write formal proofs of mathematical statements.

# Formal proofs in the wild

- Four color theorem — Appel and Haken, 1976.
- Kepler's Conjecture — Hales, 2015.
- Polynomial Freiman-Ruzsa (PFR) conjecture over $\mathbb{F}_2$ — Terrance Tao, 2023.

Prof at UCLA, Fields Medalist,

Considered greatest living mathematician



Terence Tao
@tao@mathstodon.xyz

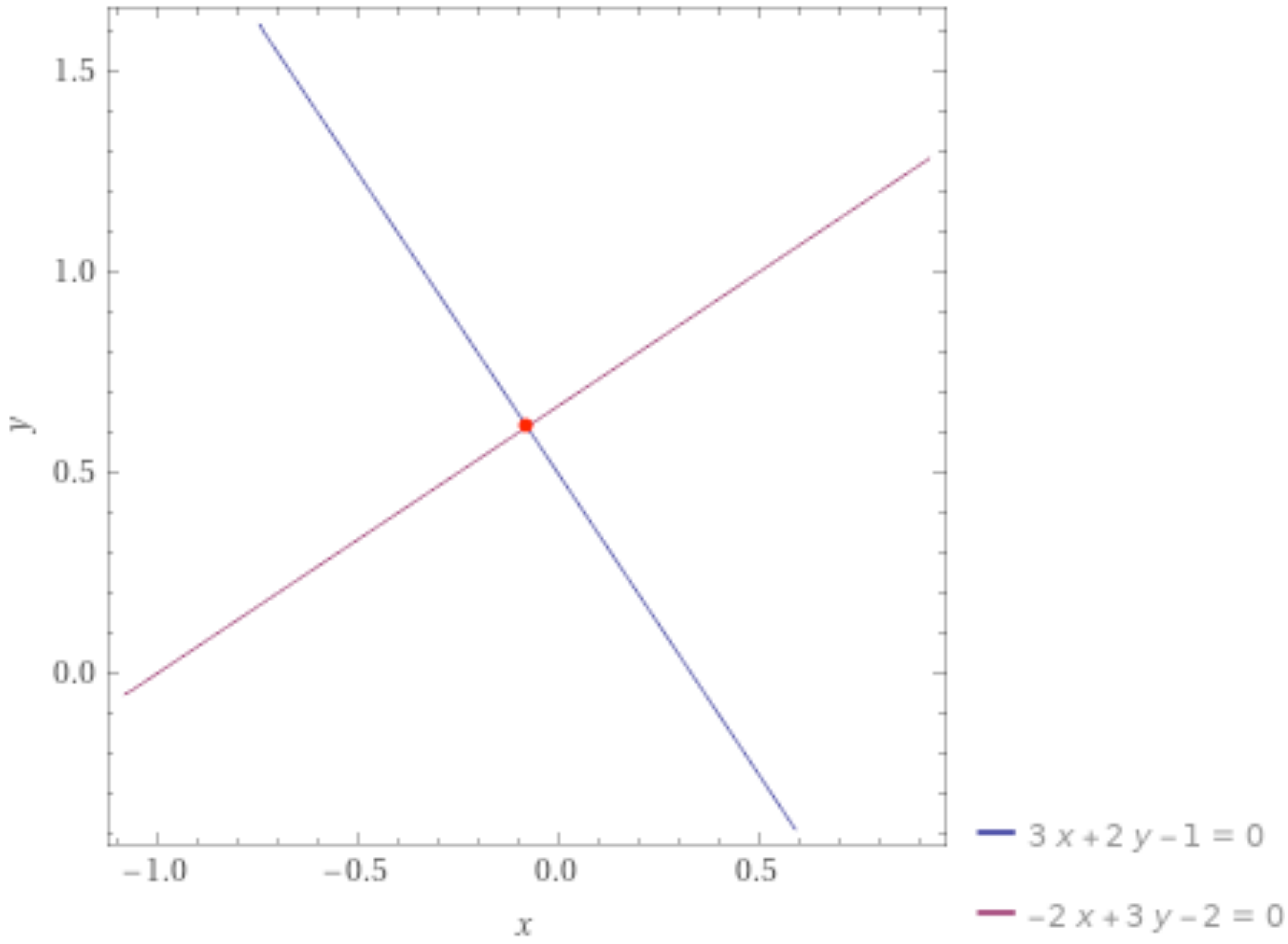
As a consequence of my #Lean4 formalization project I have found a small (but non-trivial) bug in my paper! While in the course of formalizing the arguments in page 6 of arxiv.org/pdf/2310.05328.pdf , I discovered that the expression $\frac{1}{2}\log\frac{n-1}{n-k-1}$ that appears in those arguments actually diverges in the case $n = 3, k = 2$! Fortunately this is an issue that is only present for small values of $n$, for which one can argue directly (with a worse constant), so I can fix the argument by changing some of the numerical constants on this page (the arguments here still work fine for $n \geq 8$, and the small $n$ case can be handled by cruder methods).

Enclosed is the specific point where the formalization failed; Lean asked me to establish $0 < n - 3$, but the hypothesis I had was only that $n > 2$, and so the "linarith" tactic could not obtain a contradiction from the negation of $0 < n - 3$.

# About CSCI 5535 / ECEN 5533

1. Learn to make *precise* statements and *prove* them.
2. Learn to use this skill to study *mathematical foundations* of computer programming.

# Recall High-School Algebra …



- Consider the equations:

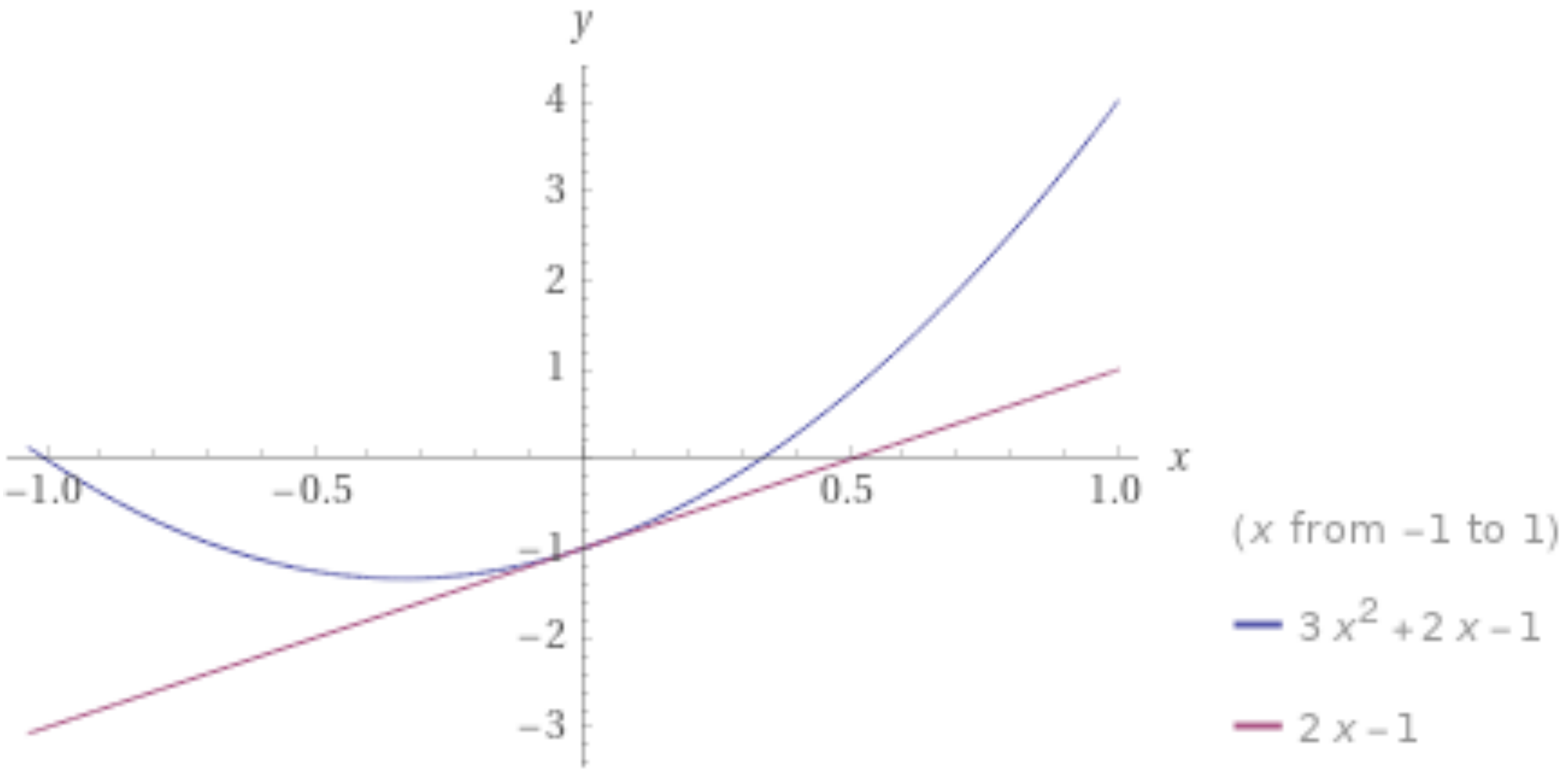
$$3x + 2y - 1 = 0$$

$$-2x + 3y - 2 = 0$$

- Different but not *fundamentally* different.

- Different instantiations of $ax + by + c = 0$

# Recall High-School Algebra ...



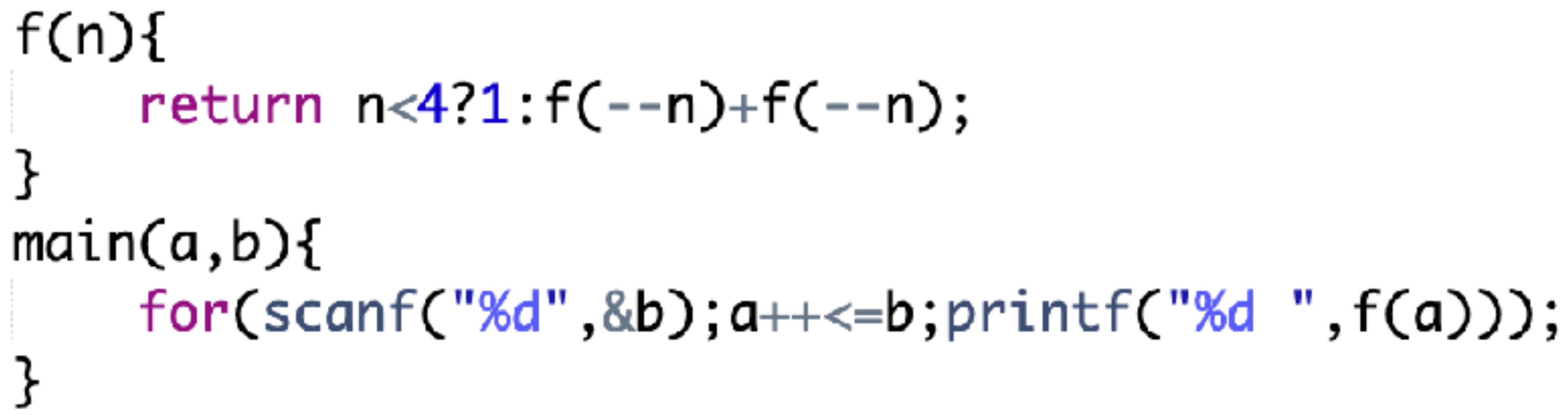
- Now consider the equations:

$$y = 3x^2 + 2x - 1$$
$$y = 2x - 1$$

- Fundamentally different equations.
  - One is quadratic, other is linear.
- $y = ax^2 + bx + c$ is more *expressive / powerful* than $ax + by + c = 0$

# Are computer programs analogous to algebraic functions?

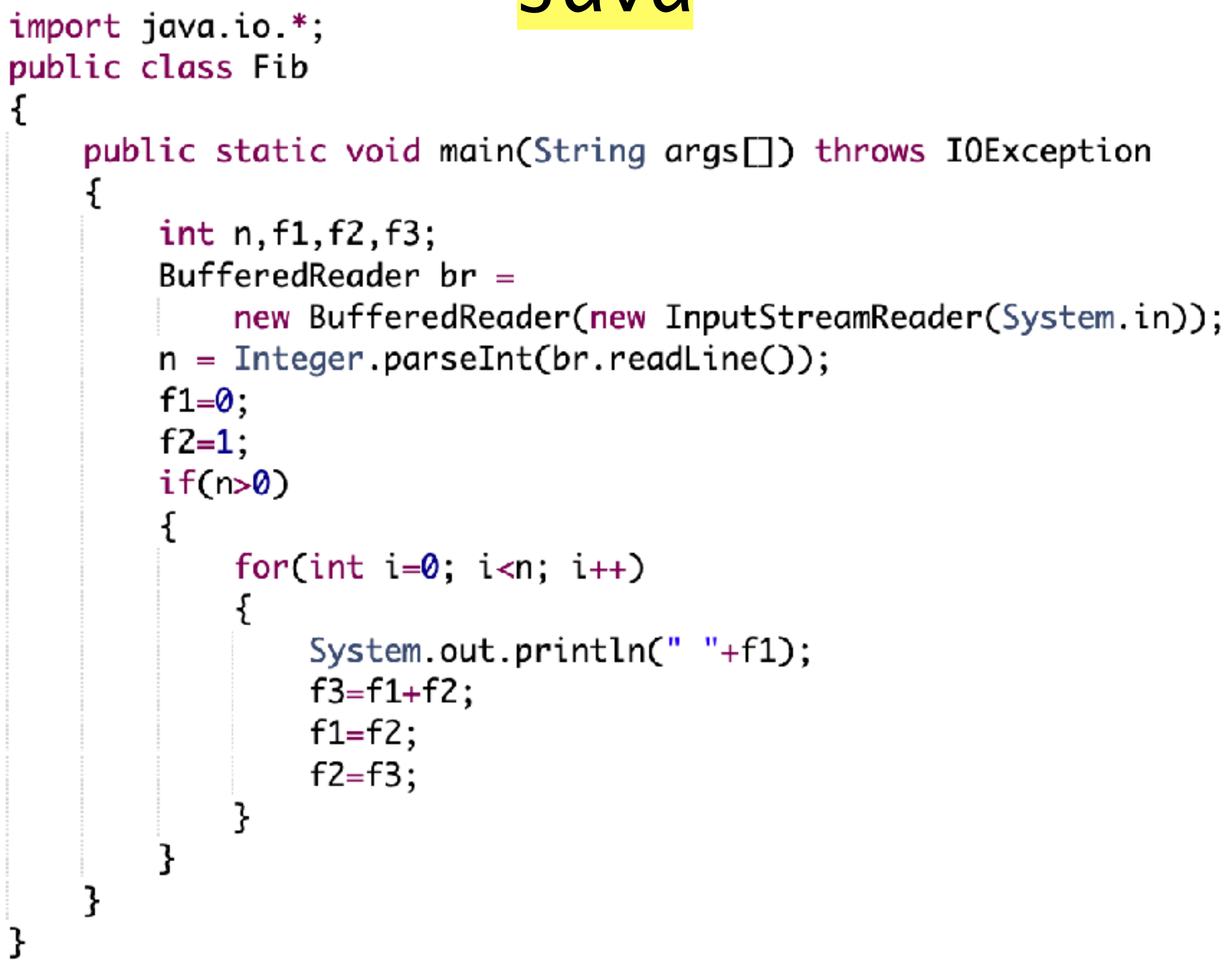
C



```c
f(n){
    return n<4?1:f(--n)+f(--n);
}
main(a,b){
    for(scanf("%d",&b);a++<=b;printf("%d ",f(a)));
}
```

$\overset{?}{\cong}$

Java



```java
import java.io.*;
public class Fib
{
    public static void main(String args[]) throws IOException
    {
        int n,f1,f2,f3;
        BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
        n = Integer.parseInt(br.readLine());
        f1=0;
        f2=1;
        if(n>0)
        {
            for(int i=0; i<n; i++)
            {
                System.out.println(" "+f1);
                f3=f1+f2;
                f1=f2;
                f2=f3;
            }
        }
    }
}
```

Q. Is there a mathematics to answer such questions decisively?

A. Yes!

# About CSCI 5535 / ECEN 5533

## Mathematical foundations of computer programs and programming languages.

- To understand fundamental differences among various programming styles and languages.
- To learn various ways in which one can ascribe a *meaning* to a program.
- To ask precise questions about computer programs and to decisively answer them.
  - E.g: “Does this program stably sort a list of numbers?”, “Does this program ever terminate?” etc.

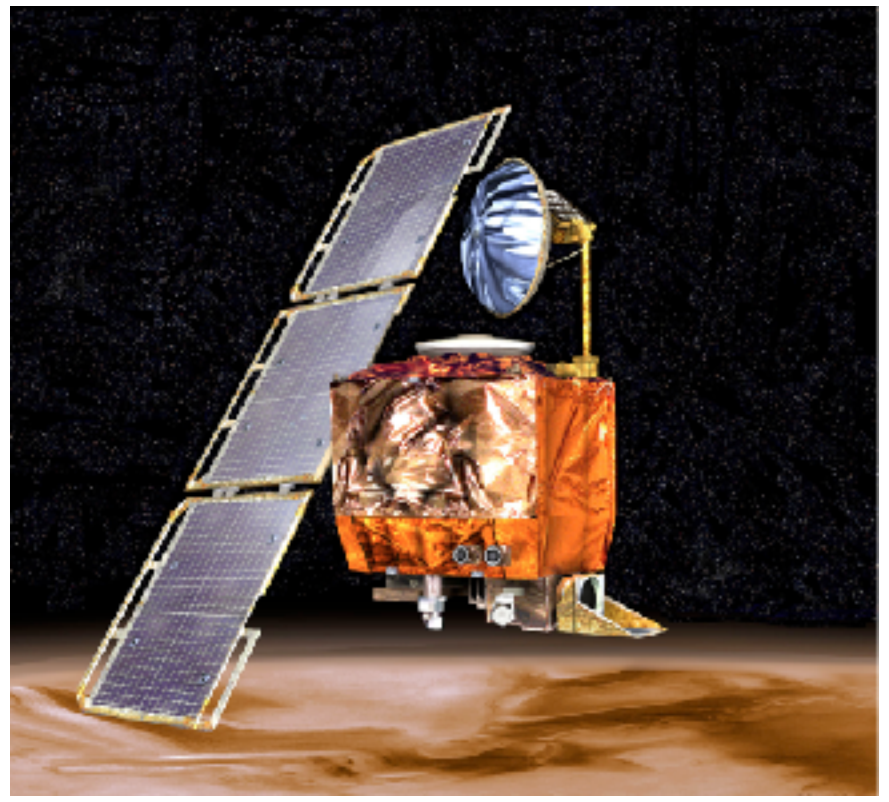
“Program Verification”

Prove that a program $P$ satisfies a property $\varphi$

# Why verify programs?

Because building reliable software is hard.

Therac 25



Mars Climate Orbiter

Boeing 737 Max 8

“*Program testing can be used to show the presence of bugs, but never to show their absence!*” - E W Dijkstra

# Does Program Verification Scale?

Use of formal methods to verify full-scale software systems is a hot research topic!

- CompCert – fully verified C compiler
  Leroy, INRIA
- Vellvm – formalized LLVM IR
  Zdancewic, Penn
- Ynot – verified DBMS, web services
  Morrisett, Harvard
- Verified Software Toolchain
  Appel, Princeton
- Bedrock – web programming, packet filters
  Chlipala, MIT
- CertiKOS – certified OS kernel
  Shao & Ford, Yale

Verified
Software
Toolchain

Bedrock

Vellvm
verified
LLVM

[Slide courtesy: B C Pierce]

# Does Program Verification Pay?

SiFive

CERTIK

amazon

galois

BAE SYSTEMS

Tezos

facebook

NASA

# Coq

Thierry Coquand

Invented Calculus of Inductive Constructions — theoretical basis for Coq

- A mechanized *proof assistant*.
  - Checks if the proof you write indeed proves the theorem you state.
- We make extensive use of Coq in this class.
- Installing Coq (version 8.12 or later):
  - From https://coq.inria.fr: You can download a Coq platform binary that includes a dedicated IDE for Coq called Coqide.
  - From https://proofgeneral.github.io: Installs a Coq major mode for Emacs. Best option if you are already familiar with Emacs.
  - Via Opam — the package manager of OCaml. See https://coq.inria.fr/opam-using.html for instructions. You can combine this with vscoq plugin for vscode: https://github.com/coq-community/vscoq.

# Evaluation Components

| Item | Count | Cumulative Weight |
|---|---|---|
| Homeworks | 6 | 48% |
| Course Project | 1 | 25% |
| Mid-term | 1 | 10% |
| Final | 1 | 15% |
| Class Participation | - | 2% |

- Coq Assignments: Write proofs in Coq for select exercises.
- 6 Homeworks: ~Once every two weeks. Submitted via Canvas. Link will be posted.
- Feel free to collaborate, but please don’t plagiarize!

# Evaluation Components

| Item | Count | Cumulative Weight |
|---|---|---|
| Homeworks | 6 | 48% |
| Course Project | 1 | 25% |
| Mid-term | 1 | 10% |
| Final | 1 | 15% |
| Class Participation | - | 2% |

- Anything related to formal verification. You chose the problem statement!
- Example 1: Formalize a model of a real-world system, and prove interesting properties.
  - Eg: Border Gateway Protocol (BGP) guarantees absence of routing loops.
- Example 2: Formally prove a landmark result from STOC/FOCS/LICS.
  - Eg: PCP theorem.
- Can be done alone or in groups of two. Expectations are scaled accordingly.

# Evaluation Components

| Item | Count | Cumulative Weight |
|---|---|---|
| Homeworks | 6 | 48% |
| Course Project | 1 | 25% |
| Mid-term | 1 | 10% |
| Final | 1 | 15% |
| Class Participation | - | 2% |

- Written exams.
- Mid-term will be in the class. Sometime in October. Date TBD.
- Final in December. Date and place determined by the registrar.
- Doing homework assignments and textbook exercises is a good practice for exams.

# TODO for you

- Checkout course website: https://csci5535.github.io
- Install Coq (v8.12 or later) and Lean4.
- Register on course Piazza (link on course website).
- Read Preface and Basics chapters from textbook Vol 1 (Logical Foundations)
- Download and run (step through) Basics.v file in your chosen Coq IDE.