# CS 565

Programming Languages (graduate)
Spring 2025

Week 1
Introduction, Functional Programming, Datatypes

# Administrivia

## Who:

**Instructor**: Suresh Jagannathan
Office Hours: MW, 11am - 12pm (LWSN 3154J)

**TA**: Songlin Jia

Office Hours: Tuesday 11 am - 12 pm (remote)

## Where: LWSN B151

## When:

January 13 - May 2, 2025
MWF 12:30pm - 1:20pm

Discussion Board: Piazza
Homeworks and Quizzes: Brightspace and Gradescope

# Structure and Grading

## Lectures

- In-person lectures

## Homeworks (35%)

- Approximately 7 over the course of the semester
- Typically 2 weeks to complete
- Involves programming and proving in Coq and Dafny

## Quizzes (10%)

- Once a week
- Short answer, multiple-choice on Gradescope
- Covers material covered in lecture

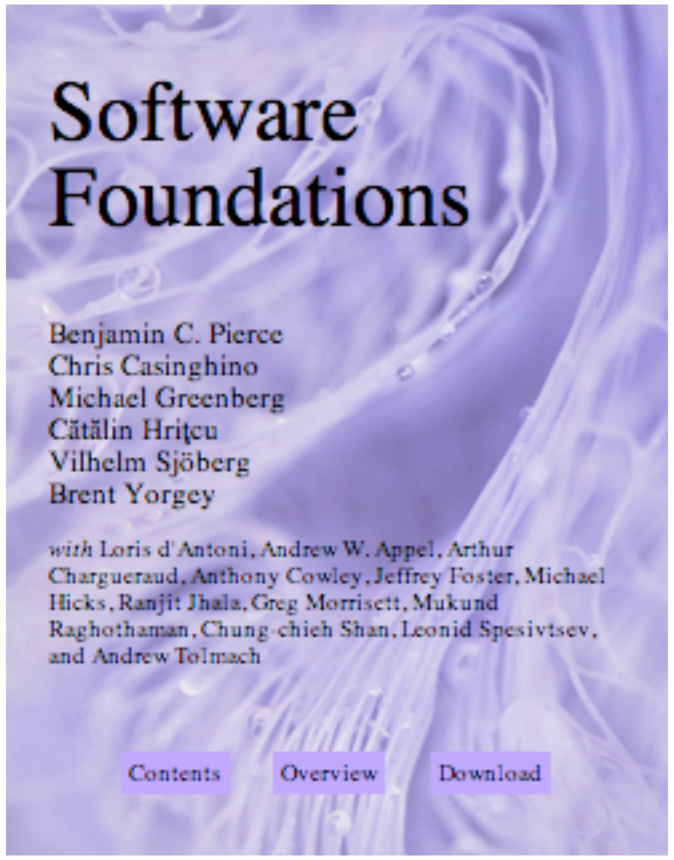
## Midterm (25%)

- Evening exam (paper)
- March 13, 8 - 9:30 PM, HAMP 1144

## Final (30%)

- Paper

# Textbooks



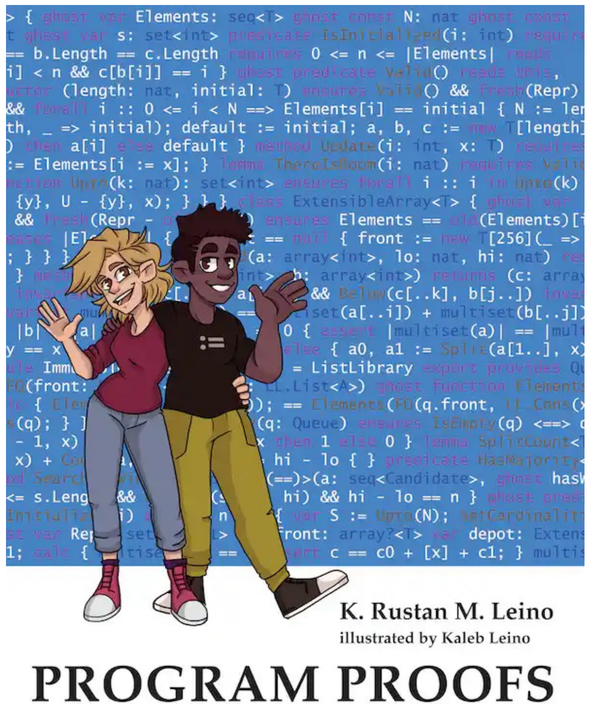
Software Foundations

Benjamin C. Pierce
Chris Casinghino
Michael Greenberg
Cătălin Hriţcu
Vilhelm Sjöberg
Brent Yorgey

*with* Loris d'Antoni, Andrew W. Appel, Arthur Charguéraud, Anthony Cowley, Jeffrey Foster, Michael Hicks, Ranjit Jhala, Greg Morrisett, Mukund Raghothaman, Chung-chieh Shan, Leonid Spesivtsev, and Andrew Tolmach

Contents Overview Download

Software Foundations



Program Proofs

## Additional Resources

- Types and Programming Languages (Pierce, 2002 MIT Press)
- Certified Programming with Dependent Types (Chlipala, eBook)

# How

to succeed
in CS 565

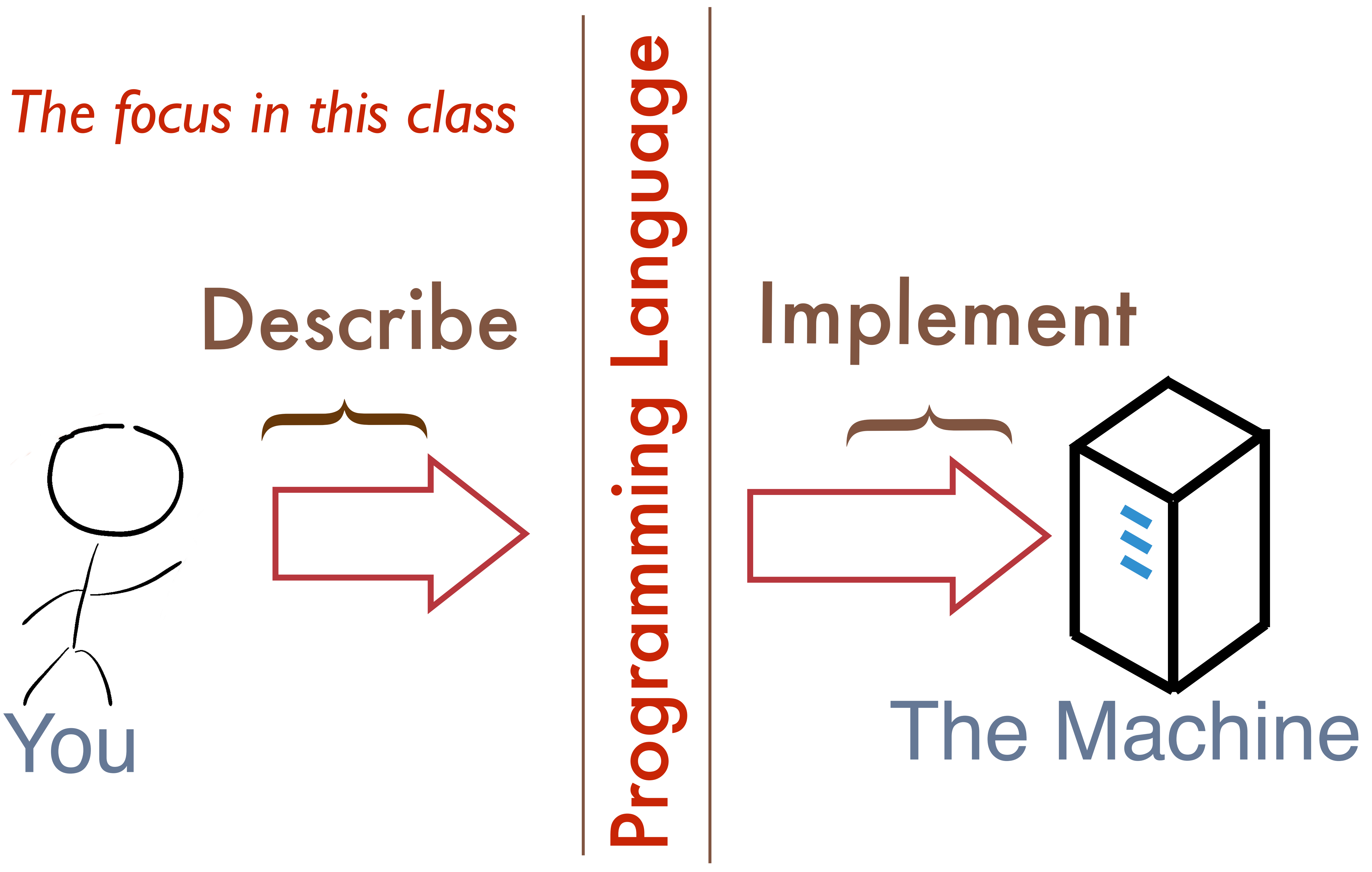
Should be familiar with:

- Programming in a high-level language
  (Python, Java, Rust, Haskell, OCaml, ...)
- Basic logic and proofs techniques
  sets, relations, functions, ...
- Basic data structures and algorithms

Participate!

Think before you prove!

# What


*The focus in this class*
Describe
Programming Language
Implement
You
The Machine

## What

Coq

### Proof Assistant:

- ★ Generate and Check Proofs
- ★ Web Page: coq.inria.fr

(Now known as Rocq)

Dafny

### Verifier-Aware Programming Language

- ★ Write programs along with specifications that are automatically verified
- ★ Web Page: dafny.org

Learning Outcome: Formalize and rigorously reason about programming languages, abstractions, and programs using these tools

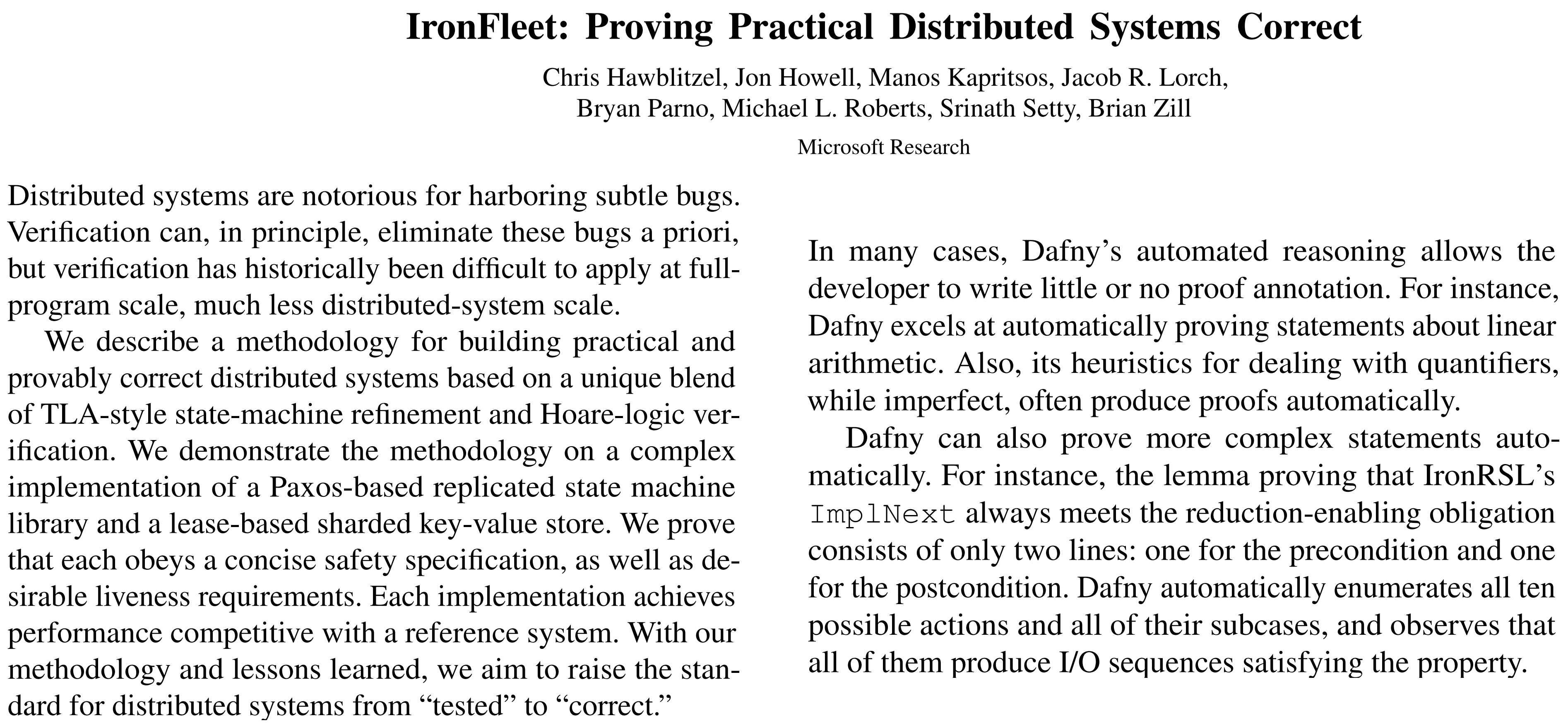
# Why?

Other Compilers: 325
CompCert: <10 (in unverified front-end)

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.
The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

Finding and Understanding Bugs in C Compilers [Yang et al. PLDI 2011]

SOSP'15



**IronFleet: Proving Practical Distributed Systems Correct**

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Microsoft Research

Distributed systems are notorious for harboring subtle bugs. Verification can, in principle, eliminate these bugs a priori, but verification has historically been difficult to apply at full-program scale, much less distributed-system scale.

We describe a methodology for building practical and provably correct distributed systems based on a unique blend of TLA-style state-machine refinement and Hoare-logic verification. We demonstrate the methodology on a complex implementation of a Paxos-based replicated state machine library and a lease-based sharded key-value store. We prove that each obeys a concise safety specification, as well as desirable liveness requirements. Each implementation achieves performance competitive with a reference system. With our methodology and lessons learned, we aim to raise the standard for distributed systems from "tested" to "correct."

In many cases, Dafny's automated reasoning allows the developer to write little or no proof annotation. For instance, Dafny excels at automatically proving statements about linear arithmetic. Also, its heuristics for dealing with quantifiers, while imperfect, often produce proofs automatically.

Dafny can also prove more complex statements automatically. For instance, the lemma proving that IronRSL's `ImplNext` always meets the reduction-enabling obligation consists of only two lines: one for the precondition and one for the postcondition. Dafny automatically enumerates all ten possible actions and all of their subcases, and observes that all of them produce I/O sequences satisfying the property.

# What

Foundations:

- ★ Functional Programming
- ★ Polymorphism and Higher-Order Programming
- ★ Propositions, Evidence, and Relations

Programming Language Semantics:

- ★ Operational Semantics
- ★ Denotational Semantics

Types:

- ★ Type Soundness
- ★ Simply-Typed Lambda Calculus, Subtyping
- ★ System F

Program Logics:

- ★ Hoare Logic (Axiomatic Semantics)
- ★ Separation Logic

Automated Program Verification

- ★ Verification-Aware Languages

# Functional Programming

We'll start our investigation by considering a small functional language

- These languages tend to have a small core set of features
- Datatypes, functions, and their application
- Written in Gallina, the specification and programming language for Coq

```
Definition double (n : nat) : nat := n + n.
```

# Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
Definition double (n : nat) : nat := n + n.

Eval compute in (double 1). (* = 2 *)
```

## Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
Definition double (n : nat) : nat :=
         plus n n.
Eval compute in (double 1). (* = 2 *)
```

## Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
Definition concat (s1 : string) (s2 : string)
                          (s3 : string) :=
      append s1 (append s2 s3).
Eval compute in (concat "Hello" " " "World").
                     (* = "Hello World" *)
```

# Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume value, produce value

```
Definition concat (s1 s2 s3 : string) : string :=
      append s1 (append s2 s3).
Eval compute in (concat "Hello" " " "World").
                  (* = "Hello World" *)
```

## Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume value, produce value
- Coq can automatically infer many type annotations

```
Definition concat s1 s2 s3 :=
      append s1 (append s2 s3).
Eval compute in (concat "Hello" " " "World").
                  (* = "Hello World" *)
```

# Building Blocks

Given the following ingredients:

- bool: a datatype for booleans
- andb: logical and
- orb: logical or
- negb: logical negation

Define a boolean equality function

```
Definition eqb (b1 b2 : bool) : bool :=
      orb (andb b1 b2) (andb (negb b1) (negb b2)).
```

# Algebraic Data Types

Enumerated types introduce nullary constructors:

```
Inductive bool : Type :=
| true : bool
| false : bool.
```

# Algebraic Data Types

- Enumerated types are the simplest data types in Coq
- Type annotations can be inferred here as well

```
Inductive bool :=
| true
| false.
```

# Algebraic Data Types

- Enumerated types are the simplest data types in Coq
- Type annotations can be inferred here
- Constructors describe how to **introduce** a value of a type

```
Inductive bool :=
| true
| false.

Inductive weekdays :=
  | monday | tuesday | wednesday | thursday | friday
: weekdays.
```

## Pattern Matching

- Pattern matching lets a program use values of a type
- Coq only permits **total** functions
    - A total function is defined on all values in its domain

```
Definition negb (b : bool) : bool :=
  match b with
  | true => false
  | false => true
  end.

Eval compute in (negb true).  (* = false *)
```

# Pattern Matching

- Pattern matching lets a program use values of a type
- Coq only permits **total** functions
    - A total function is defined on all values in its domain

```
Definition eqb (b1 b2 : bool) : bool :=
  match b1, b2 with
  | true, true => true
  | false, false => true
  | false, true => false
  | true, false => false
  end.
```

# Pattern Matching

- Pattern matching lets a program use values of a type
- Coq only permits **total** functions
    - A total function is defined on all values in its domain
- Underscores are the wildcard pattern (don’t care)

```
Definition eqb (b1 b2 : bool) : bool :=
  match b1, b2 with
  | true, true => true
  | false, false => true
  | _, _ => false
  end.
```

# Compound ADTs

- Can build new ADTs from existing ones:
  - A color is either black, white, or a primary color
  - Need to apply primary to something of type rgb
- ADTs are **algebraic** because they are built from a small set of operators (sums of product).

```
Inductive rgb : Type := | red | green | blue.

Inductive color := | black | white
   | primary (p : rgb).

Eval compute in (primary red). (* = primary red *)
```

# Pattern Matching²

- Patterns on compound types need to mention arguments
    - Can be a **variable**

```
Definition monochrome (c : color) : bool :=
  match c with
  | black => true
  | white => true
  | primary p => false
  end.
```

# Pattern Matching$^2$

- Patterns on compound types need to mention arguments
    - Can be a **variable**
    - Can be a **pattern** for the type of the argument

```
Definition isred (c : color) : bool :=
  match c with
  | black => false
  | white => false
  | primary red => true
  | primary _ => false
  end.
```

# Concept Check

- How many colors are there?
- In general, each ADT defines an algebra whose operations are the constructors

```
Inductive rgb : Type := | red | green | blue.

Inductive color := | black | white
  | primary (p : rgb).

Eval compute in (primary red). (* = primary red *)
```

## Concept Check$^2$

- Define a type for the 'basic' (h, a, and p) html tags:
  - A header should include a nat indicating its importance
  - The anchor tag should include a string for its destination
  - The paragraph doesn't need anything extra

```
Inductive tag : Type :=
| h (importance : nat)
| a (href : string)
| p.
```

# Concept Check$^2$

- Define a pretty printer for opening a tag

(* pp (h 1) = “<h1>” *) *)

- Assume we have a natToString function

```
Inductive tag : Type :=
| h (importance : nat)
| a (href : string)
| p.
```

## Concept Check$^2$

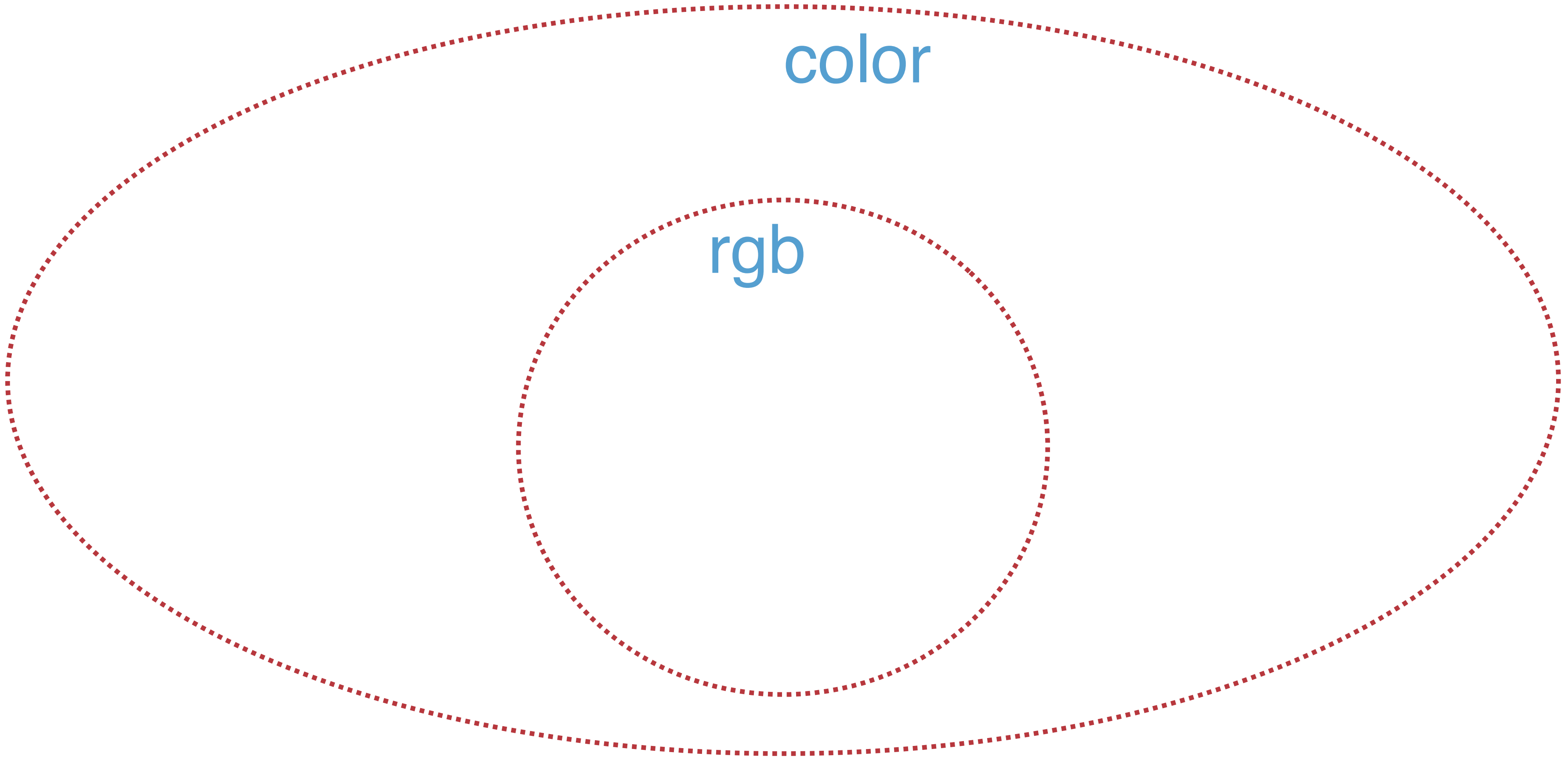
- Define a pretty printer for opening a tag
  - (* pp (h 1) = “<h1>” *) *)
  - Assume we have a natToString function

```
Definition pp (t : tag) : string :=
  match t with
  | h i => concat "<h" (natToString i) “>”
  | a hr => concat "<a href='" hr "'>"
  | _ => “<p>"
  end.
```

# So Far:

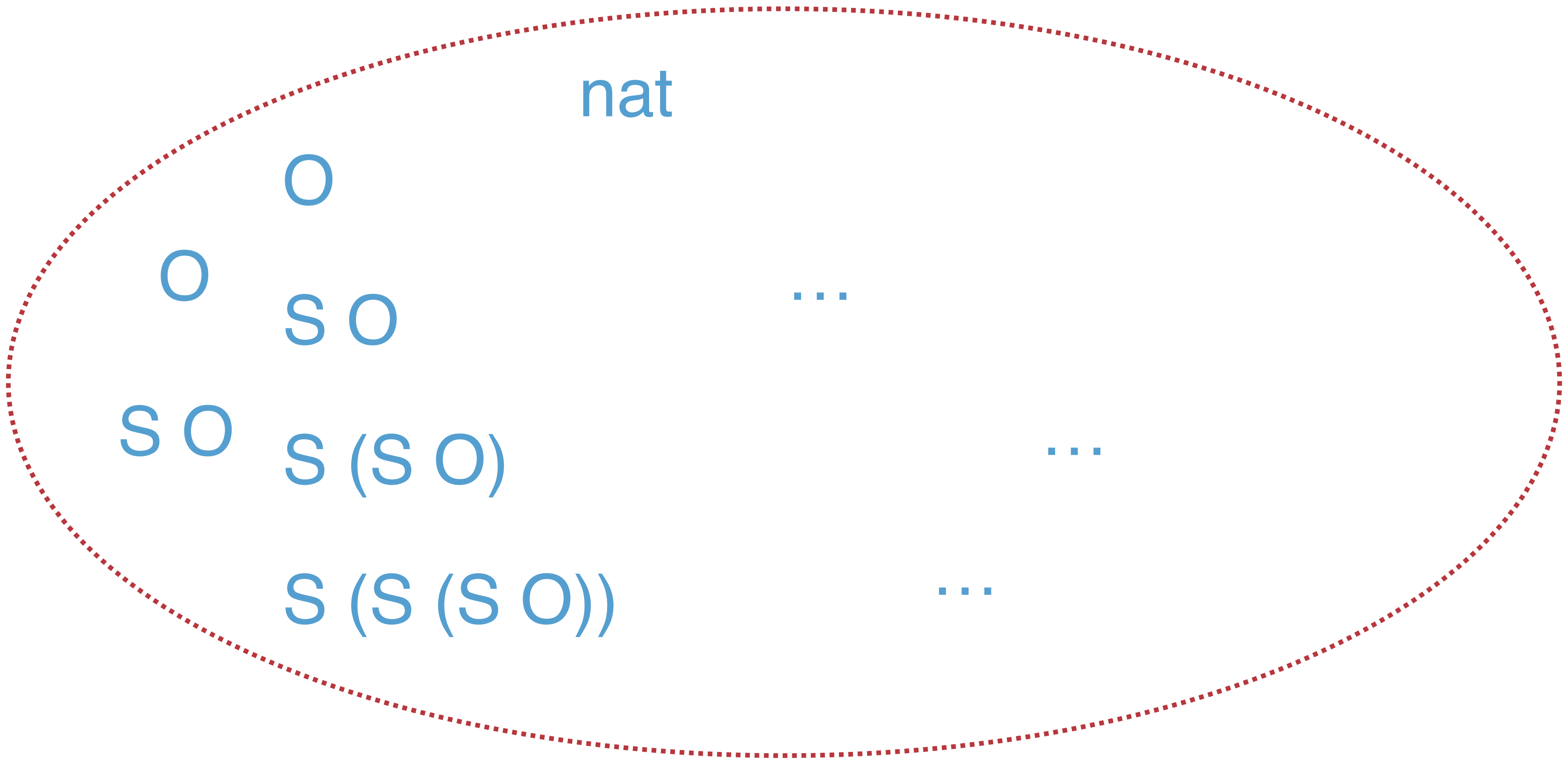
```
Inductive rgb : Type := | red | green | blue.

Inductive color := | black | white
  | primary (p : rgb).
```



# Natural Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```



# Functions

The *interpretation* of these constructors comes from how we use them to compute:

```
Inductive tickNat : Type :=
   | stop
   | tick (foo : tickNat).
```

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S m => m
  end.
```

## Recursion

Recursive functions use themselves in their definition

```
Fixpoint iseven (n : nat) : bool :=
???
```

# Recursion

Recursive functions use themselves in their definition

```
Fixpoint iseven (n : nat) : bool :=
  match n with
  | O => true
  | S O => false
  | S (S m) => iseven m
  end.
```

## Recursion

Recursive functions use themselves in their definition

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O => m
  | S n’ => S (plus n’ m)
  end.
Eval compute in (plus 2 3). (* = 5 *)
```

## Recursion

Recursive functions use themselves in their definition

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O => m
  | S n’ => S (plus n’ m)
  end.
Eval compute in (plus 2 3). (* = 5 *)
(* plus 2 5 = plus (S (S O)) (S (S (S O))) *)
```

# Recursion

Recursive functions use themselves in their definition

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O => m
  | S n’ => S (plus n’ m)
  end.
Eval compute in (plus 2 3). (* = 5 *)
(* plus (S (S O)) (S (S (S O))) =
      S (plus (S O) (S (S (S O))))*)
```

# Recursion

Recursive functions use themselves in their definition

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O => m
  | S n’ => S (plus n’ m)
  end.
Eval compute in (plus 2 3). (* = 5 *)
(* S (plus (S O) (S (S (S O)))) =
      S (S (plus O (S (S (S O)))))*)
```

## Recursion

- Recursive functions use themselves in their definition
- Recall: functions need to be **total**
  - Coq requires functions be structurally recursive

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O => m
  | S n’ => S (plus n’ m)
  end.
Eval compute in (plus 2 3). (* = 5 *)
(* S (S (plus O (S (S (S O))))) =
      S (S (S (S (S O)))) = 5 *)
```

# Recursion

- ★ Recursive functions use themselves in their definition
- ★ Recall: functions need to be **total**
  - ★ Coq requires functions be structurally recursive

```
Fixpoint mult (n m : nat) : nat :=
  match n with
  | O => O
  | S n' => plus m (mult n' m)
  end.
```

# Recursion

- Recursive functions use themselves in their definition
- Recall: functions need to be **total**
  - Coq requires functions be structurally recursive

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus m n’)
  end.
```

## Putting it together: Syntax

(of Arithmetic + Boolean Expressions)

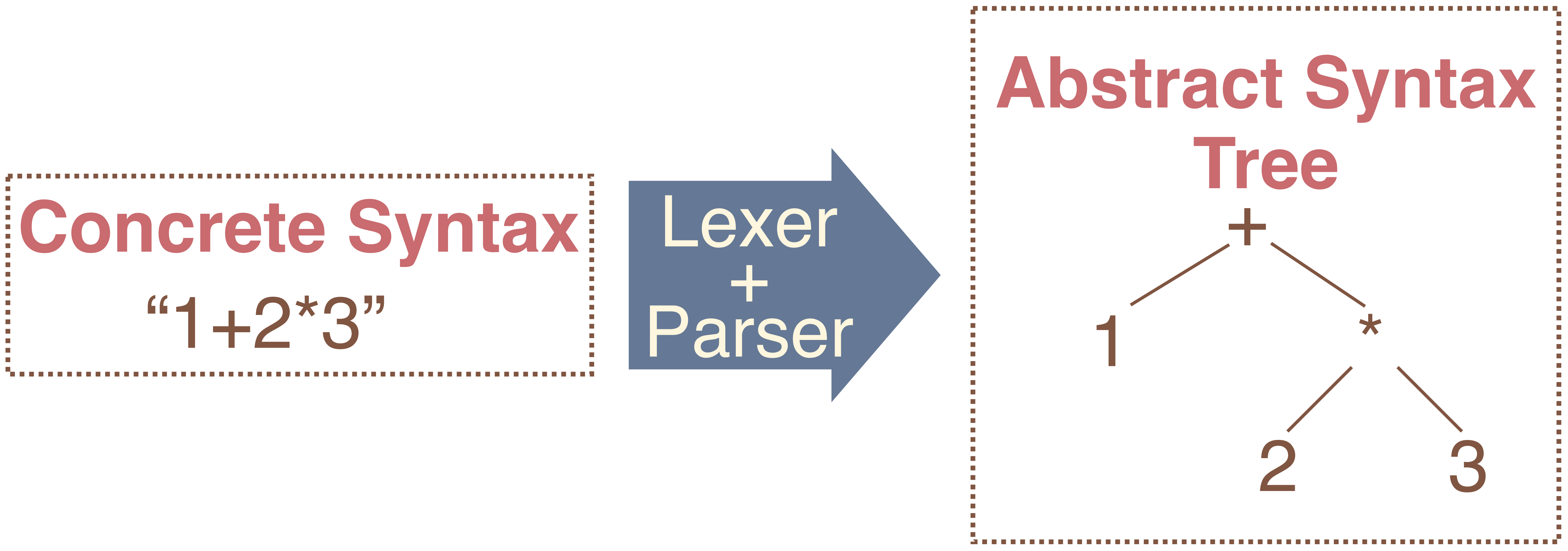
Backus-Naur Form (BNF) Definitions:

```
A ::= ℕ
    | A + A
    | A - A
    | A * A
```

```
B ::= true
    | false
    | A = A
    | A ≤ A
    | ¬ B
    | B ∧ B
```

# Abstract Syntax

## (of Arithmetic + Boolean Expressions)



# Syntax in Coq

```
A ::= ℕ
    | A + A
    | A - A
    | A * A
```

```
Inductive aexp : Type :=
  | ANum (a : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult  (a1 a2 : aexp).
```

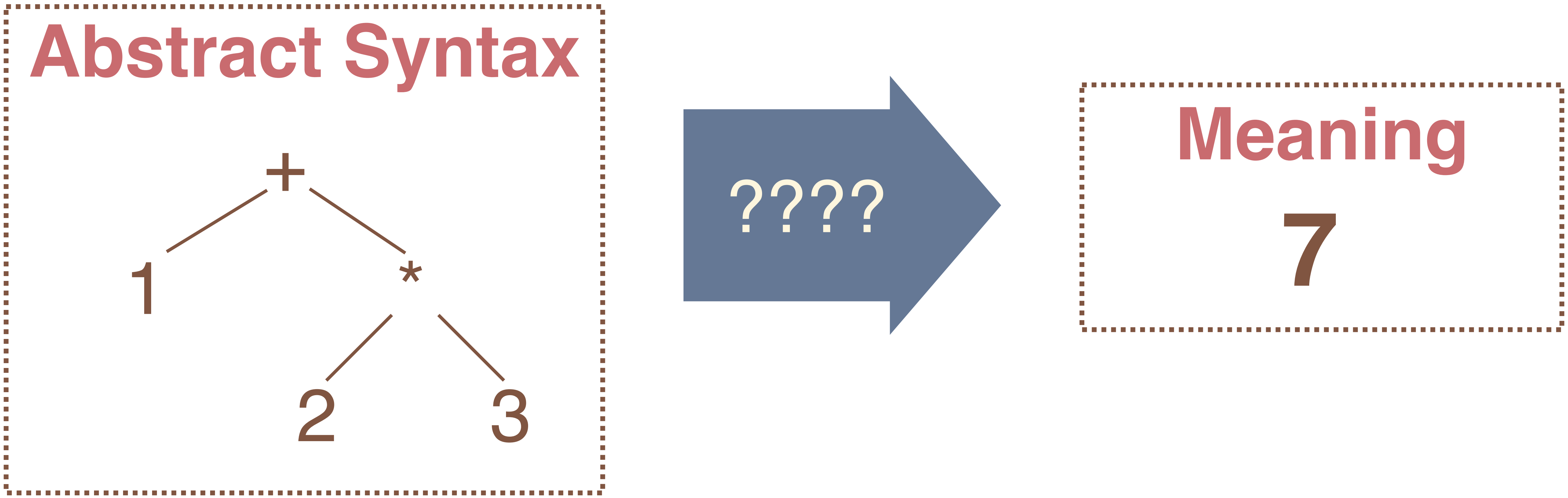
- ★ One constructor per rule
- ★ Nonterminal = inductive type being defined

# Syntax in Coq

```
B ::= true
    | false
    | A = A
    | A ≤ A
    | ¬ B
    | B ∧ B
```

```
Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).
```

# Evaluation



# Evaluation

- ★ The evaluator for axep is simply a recursive function

```
Fixpoint aeval (a : aexp) :  (* ?? *)  :=
  match a with
  | ANum n => n
  | APlus a1 a2 => (aeval a1) + (aeval a2)
  | AMinus a1 a2 => (aeval a1) - (aeval a2)
  | AMult a1 a2 => (aeval a1) * (aeval a2)
  end.
```

# Evaluation

★ The evaluator for axep is simply a recursive function

```
Fixpoint aeval (a : aexp) :  nat  :=
  match a with
  | ANum n => n
  | APlus a1 a2 => (aeval a1) + (aeval a2)
  | AMinus a1 a2 => (aeval a1) - (aeval a2)
  | AMult a1 a2 => (aeval a1) * (aeval a2)
  end.
```

# Evaluation

★ An evaluator for boolean expressions

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue => true
  | BFalse => false
  | BEq a1 a2 => eqb (aeval a1) (aeval a2)
  | BLe a1 a2 => leb (aeval a1) (aeval a2)
  | BNot b => negb (beval b)
  | BAnd b1 b2 => andb (beval b1) (beval b2)
  end.
```