CS 565

Programming Languages (graduate)
Spring 2026

Week 1

Introduction, Background, Functional Programming

Administrivia

- Slides posted on course webpage

  https://www.cs.purdue.edu/homes/suresh/565-Spring2026
- See course page for details on
  - grading
  - exams
  - syllabus

Topics

Foundations:

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

Programming Language Semantics:

- Operational Semantics

Types:

- Simple Types
- Simply-Typed Lambda Calculus
- Subtyping
- References and Linear/Affine Types
- System F

Program Logics:

- Hoare Logic (Axiomatic Semantics)
- Separation Logic

Automated Program Verification

- Verification-Aware Languages (Dafny)

Preliminaries

Sets: Basic Concepts

- Sets as collections of distinct elements
- Membership: $x \in A$

  Examples: $\{1,2,3\}$, $\varnothing$, $\{x \in \mathbb{N} \mid x \text{ is even}\}$

Operations

- Union $(A \cup B)$, Intersection $(A \cap B)$
- Difference $(A \setminus B)$

  Example: $\{1,2,3\} \cup \{3,4\} = \{1,2,3,4\}$

- Cartesian Product: $A \times B = \{(a,b) \mid a \in A, b \in B\}$

  Example: $\{1,2\} \times \{a,b\} = \{(1,a),(1,b),(2,a),(2,b)\}$

Preliminaries

Power Sets and Lattices

- $\mathcal{P}$(A) = all subsets of A
- Example: $\mathcal{P}$({a,b}) = {∅,{a},{b},{a,b}}
- Ordered by ⊆, forms a complete lattice

```
let powerset (xs : 'a list) : 'a list list =
  match xs with
    | [] -> [ [] ]
    | x :: rest ->
        let ps = powerset rest in
        ps @ List.map (fun s -> x :: s) ps
```

Preliminaries

- A partially ordered set $(L, \leq)$ is a *lattice* if every pair $(x,y) \in L$ has:
  - a least upper bound (join): $x \vee y$
  - a greatest lower bound (meet): $x \wedge y$
  - Joins and meets are unique
- A complete lattice is a poset (partially-ordered set) where every subset $X \subseteq L$ has:
  - a supremum ($\vee X$) – least upper bound
  - an infimum ($\wedge X$) – greatest lower bound
  - Includes infinite joins and meets

<u>Example:</u>

Let A be any set

- $(\mathcal{P}(A), \subseteq)$ forms a complete lattice
- Join (supremum): $\vee X = \cup X$, Meet (infimum): $\wedge X = \cap X$
- Bottom element: $\varnothing$; Top element: A

Preliminaries

Functions and Relations

- Function $f : A \rightarrow B$ assigns one output in B per input in A
  - Total vs partial functions
    - Injective: $f(x)=f(y) \Rightarrow x = y$
    - Surjective: every $b \in B$ has a preimage
    - Bijective: both
  - Image: $f(S) = \{f(x) \mid x \in S\}$
  - Pre-image: $f^{-1}(T) = \{x \mid f(x) \in T\}$

- Relation $R \subseteq A \times B$, Examples: $=$, $\leq$, $\rightarrow$
  - All functions are relations
  - Different kinds of relations:
    - reflexive, symmetric, transitive (equivalence)
    - reflexive, asymmetric, transitive (partial order)

Fixpoints

- Given a function f : L → L, a fixpoint x satisfies f(x) = x
  - Fixpoints represent stable meanings of recursive definitions
  - There may be many fixpoints in a lattice
  - Least fixpoint is usually of interest in semantics

- Kleene fixpoint theorem
  - Let (L, ≤) be a complete lattice
  - If f : L → L is monotone (i.e., $X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$) then:
    * f has a least fixpoint
    * $lfp(f) = \vee \{ \bot, f(\bot), f^2(\bot), \ldots \}$
    * Constructed by iterating from bottom

Preliminaries

Fixpoint example

Let $A = \{a, b, c\}$

- Consider the complete lattice $(\mathcal{P}(A), \subseteq)$
- Define $f(X) = X \cup \{a\}$
- Monotonicity: $X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$
- Kleene iteration: $\varnothing \subseteq \{a\} \subseteq \{a\} \subseteq \ldots$
- Least fixpoint: $\{a\}$

Preliminaries

Logic

- Propositional
  - Connectives: $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$
    - Truth tables define semantics
    - Example: $(P \wedge Q) \rightarrow P$
- Predicate
  - Quantifiers $\forall$ and $\exists$
    - Predicates over domains
    - Example: $\forall n \in \mathbb{N}.\ n+1>n$

Lambda Calculus

- Lambda calculus was developed by Alonzo Church in the 30s
  - A core language in which *everything* is a function
- Syntax of Lambda terms:

```
t ::= x
    | λx.t
    | t t
```

Variable

Lambda abstraction

Application

Lambda Calculus


```
t ::= x
    | λx.t
    | t t

x ∈ Var
```

Identity function:

λx.x

Lambda Calculus

PLUS NUMBERS

```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

Identity function:

`λx.x`

Double function:

`λx.x + x`

Applying a function:

`(λx.x) 42`

Lambda Calculus

PLUS NUMBERS


```
t ::= x
     | λx.t
     | t t
     | n
     | t + t
x ∈ Var
n ∈ N
```

Identity function:

λx.x

Double function:

λx.x + x

Applying a function:

(λx.x) (λx.x)

Lambda Calculus

PLUS NUMBERS


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

Identity function:

```
λx.x
```

Double function:

```
λx.x + x
```

Applying a function:

```
(λx.λy.x) (λx.x)
```

Lambda Calculus

Plus Numbers


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

Identity function:

```
fun x -> x
```

Double function:

```
fun x -> x + x
```

Applying a function:

```
(fun x -> x) 42
```

Conventions


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

- Application associates to the left:

  `s t u ≡ (s t) u`

- Group sequences of lambda abstractions:

  `λx y. x ≡ λx. λy. x`

- Bodies of abstraction extend as far to the right as possible:

  `λx y. x y x ≡`
  `λx.(λy.((x y) x))`

Variable Scopes


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

1. A variable x is **bound** when it occurs in the body t of a lambda abstraction `λx.t`:

2. A variable x is **free** if it is not bound by an enclosing lambda expression:

3. A **closed** term has no free variables

Concept Check

What the **free** and **bound** variables in these terms?

- `λx.λy.y x z`
- `(λx.λy.y x) (5+2) λx.x+1`
- (λx.x) (λx.x y) (λz.(λy.y) z)

α-Equivalence


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

1. Variables are bound to the closest enclosing lambda:

2. The name of bound variables is not important:

3. Expressions t_1 and t_2 that differ only in bound variable names are called **α-equivalent**

Concept Check

Which of these terms are **α-equivalent?**

$(\lambda x.x)\ ((\lambda w.w)\ ((\lambda z.(\lambda y.y)\ z)) \equiv_\alpha (\lambda x.x)\ ((\lambda x.x)\ ((\lambda x.(\lambda x.x)\ x))$

$(\lambda x.\lambda y.y\ x)\ (5+2)\ \lambda x.x+1 \equiv_\alpha (\lambda q.\lambda y.y\ q)\ (5+2)\ (\lambda y.y+1)$

$(\lambda x.\lambda y.y\ x)(5+2)\lambda x.x+1 \equiv_\alpha ((\lambda q.\lambda y.y\ q)(5+2))\ (\lambda x.x+1)$

$(\lambda x.\lambda y.y\ x)\ (5+2)\ \lambda x.x+1 \equiv_\alpha (\lambda x.\lambda y.y\ x)\ 7\ \lambda x.x+1$

$(\lambda x.\lambda y.y\ x\ z) \equiv_\alpha (\lambda a.\lambda b.b\ c\ z)$

$(\lambda y.\lambda x.x\ y\ q) \equiv_\alpha (\lambda x.\lambda y.y\ x\ z)$

Inference Rules

To describe the meaning of lambda-calculus expressions, we will use a notation called *inference (or reduction) rules*.

Informally, a rule of the form:

$$\frac{A_1, A_2, \ldots, A_n}{\mathtt{t}_1 \rightarrow \mathtt{t}_2}$$

reads:

Expression $\mathtt{t}_1$ evaluates to (or "reduces" to) $\mathtt{t}_2$
if the constraints defined by $A_1, A_2, \ldots, A_n$ hold

We'll delve into a more formal characterization of what these rules signify later in the course …

Semantics

REDUCTION RULES

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\frac{\text{value } t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{\text{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x{\coloneqq}t_2]t_1}$$

Read $[x{\coloneqq}t_2]t_1$ as "replace all free occurrences of x in t_1 with t_2"

This rule is called the **beta reduction rule**

VALUE RULES

$$\frac{}{\text{value } (\lambda x.t)}$$

Semantics

Plus Numbers

Reduction Rules

$$\frac{\texttt{value } t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1 + t_2 \longrightarrow t_1 + t_2}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}$$

$$\frac{\texttt{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1}$$

$$\frac{n \in \mathbb{Z} \qquad m \in \mathbb{Z}}{n + m \longrightarrow n +_{\mathbb{Z}} m}$$

Value Rules

$$\frac{}{\texttt{value } (\lambda x.t)}$$

$$\frac{n \in \mathbb{Z}}{\texttt{value } n}$$

Substitution

Need to ensure that we don't inadvertently bind free variables!

$$[x{:=}s]x \equiv s$$
$$[x{:=}s]y \equiv y \quad \text{if } x \neq y$$
$$[x{:=}s]\lambda x.t \equiv \lambda x.t$$
$$[x{:=}s]\lambda y.t \equiv \lambda y.[x{:=}s]t \quad \text{where } x \neq y$$
$$[x{:=}s]t_1\ t_2 \equiv [x{:=}s]t_1\ [x{:=}s]t_2$$

Not sufficient when s is an open term

$$[x{:=}w](\lambda y.x) \equiv \lambda y.\ w$$
$$[x{:=}\lambda z.z\ w](\lambda y.x) \equiv \lambda y\ z.\ z\ w$$
$$[x{:=}y](\lambda x.x) \equiv \lambda x.x$$
$$[x{:=}w\ y\ z](\lambda z.x\ z) \equiv \lambda z.(w\ y\ z)\ z$$

$$[x{:=}w\ y\ z](\lambda z.x\ z) \not\equiv \lambda z.(w\ y\ z)\ z$$
$$\equiv_\alpha [x{:=}w\ y\ z](\lambda u.x\ u) \equiv \lambda u.(w\ y\ z)\ u$$

Semantics

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\frac{\text{value } t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{\text{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1}$$

β-redex → $(\lambda x.t_1)$

Redexes are highlighted in blue

$(\lambda x.\ \lambda y.\ x\ y)\ (\lambda z.\ z)\ (\lambda w.\ w) \longrightarrow$

$(\lambda y.\ (\lambda z.\ z)\ y)\ (\lambda w.\ w) \longrightarrow$

$(\lambda z.\ z)\ (\lambda w.\ w) \longrightarrow$

$(\lambda w.\ w)$

A term with no redexes is said to be in **normal form**

Example

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\frac{\text{value } t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{\text{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x{:=}t_2]t_1}$$

```
(λx.x) (λx. x (λt f. f) (λt f. t)) (λt f. t)
 ⟶ (λx. x (λt f. f) (λt f. t)) (λt f. t)
 ⟶ (λt f. t) (λt f. f) (λt f. t)
 ⟶ (λf. (λt f. f)) (λt f. t)
 ⟶ λt f. f
```

Concept Check

Identify any redexes in the following terms:

```
(λx.x) (λx.x)

λz.(λx.x) z

(λx.x) ((λy.y) (λz.(λx.x) z))

λx y. x y x
```

Evaluation Strategies

CALL-BY-VALUE AKA STRICT

Recall that lambda abstractions and numbers are values:

The lambda calculus' values are the functions:

$$\frac{}{\text{value } \lambda x.t}$$

This is called a *call-by-value* semantics: redexes are always the top-most function that is applied to a value:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\frac{\text{value } t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{\text{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x:=t_2]t_1}$$

Examples


```
(λx. x + x) ((λx. x + x) (5 + 3)) ⟶
    (λx. x + x)((λx. x + x) 8) ⟶
        (λx. x + x)(8 + 8) ⟶
         ((λx. x + x) 16) ⟶
              16 + 16 ⟶
                  32
```

```
(λx.λy.y x)(5+2) λx.x+1
⟶(λx.λy.y x) 7 λx.x+1
⟶(λy. y 7) λx.x+1
⟶(λx. x+1) 7
⟶ 7+1
⟶ 8
```

Normalization

- If *every* program in a language is guaranteed to always evaluate to a normal term, we say the language is *strongly normalizing*.
  - Formally:
  - **Statement of Strong Normalization:**
  - For any term `t`, all sequences of reduction steps starting from `t` eventually reaches a normal form `t'`.
- Every program in a strongly normalizing language terminates.

Ω

- Is the lambda calculus strongly normalizing under beta reduction?
  - Does every expression eventually evaluate to a normal form?
  - No!

This is a diverging computation, i.e. one that does not terminate
We'll call this Ω

$$\Omega \equiv (\lambda\ \mathrm{x}.\ (\mathrm{x}\ \mathrm{x}))(\lambda\ \mathrm{x}.\ (\mathrm{x}\ \mathrm{x}))$$

Evaluation Strategies

CALL-BY-NAME
AKA LAZY

An alternative: beta-reductions are performed as soon as possible:

$$\frac{}{(\lambda x.t_1)\ t_2 \longrightarrow [x \coloneqq t_2]t_1}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

```
(λx.λy.y x)(5+2)λx.x+1
⟶(λy.y (5+2)) λx.x+1
⟶(λx.x+1) (5+2)
⟶(5 + 2) + 1
⟶ 7 + 1
⟶ 8
```

```
(λf.f 7)((λx.x x) λy.y)
⟶ ((λy. y) (λy. y)) 7
⟶ (λy. y) 7
⟶ 7
```

term duplicated!

Evaluation Strategies

Call-By-Name

$(\lambda x.x + x)(5 + 6)$
$\longrightarrow (5 + 6) + (5 + 6)$
$\longrightarrow 11 + (5 + 6)$
$\longrightarrow 11 + 11$
$\longrightarrow 22$

Laziness can lead to duplicated work!

Call-By-Value

$(\lambda x\ y.x + x)\ 5\ (5 + 6)$
$\longrightarrow (\lambda y.5 + 5)\ (5 + 6)$
$\longrightarrow (\lambda y.5 + 5)\ 11$
$\longrightarrow 5 + 5$
$\longrightarrow 10$

Strictness can lead to unnecessary work!

Concept Check

Evaluate this expression using both CBV and CBN strategies:

```
(λx.x) ((λy.y) (λz.(λx.x) z))
```

(Recall application is left-associative)

Eta-reduction

One common additional reduction rule is called **eta reduction**:

$$\frac{\texttt{x does not appear in t}}{(\lambda\texttt{x.t x}) \longrightarrow \texttt{t}}$$

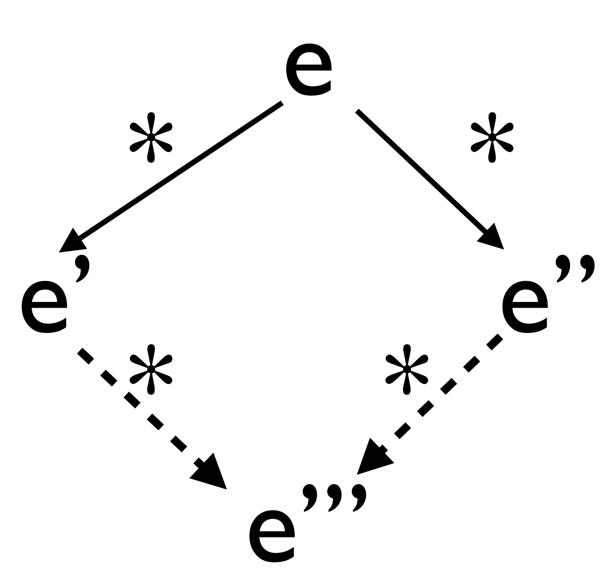
Captures the idea that λx. (λy.y x) and λy.y are equivalent

Properties

Church-Rosser Theorem (1)

If e $\longrightarrow^*$ e' and e $\longrightarrow^*$ e'' then there exists a term e''' such that e' $\longrightarrow^*$ e''' and e'' $\longrightarrow^*$ e'''

(Here $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$)



- The reduction rules of the lambda calculus are confluent
- Normal forms are unique

Properties

Church-Rosser II

A reduction strategy that always reduces the leftmost, outermost redex of a term will yield a normal form, if it exists.

- A call-by-name evaluation strategy guarantees reduction to normal form (if it exists)
- This property does not hold under by call-by-value. Why?

Expressivity

Church's Thesis (1935): Informally, any function on the natural numbers that can be effectively computed (i.e., can be expressed as an algorithm) can be computed using the λ-calculus. In other words, λ-calculus is equivalent in its expressive power to Turing Machines.

- This property holds for the pure λ-calculus, i.e., the calculus without primitive support for numbers!
- This means that function abstraction and application are sufficiently powerful to model numbers and their operations.

Functional Programming

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

- These languages tend to have a small core set of features
    * Based on lambda-calculus
- Extend this core with
    * algebraic datatypes
    * primitive support for recursion
    * pattern-matching and conditionals
    * strong typing
    * syntactic sugar
- Written in Gallina, the specification and programming language for Rocq

```
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
- Rocq 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 Rocq
- Type annotations can be inferred here as well

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

Algebraic Data Types

- Enumerated types are the simplest data types in Rocq
- 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
- Rocq 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
- Rocq 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
- Rocq 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 Matching[2]

- 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 Matching²

- 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 Check²

- 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 Check²

- 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 Check²

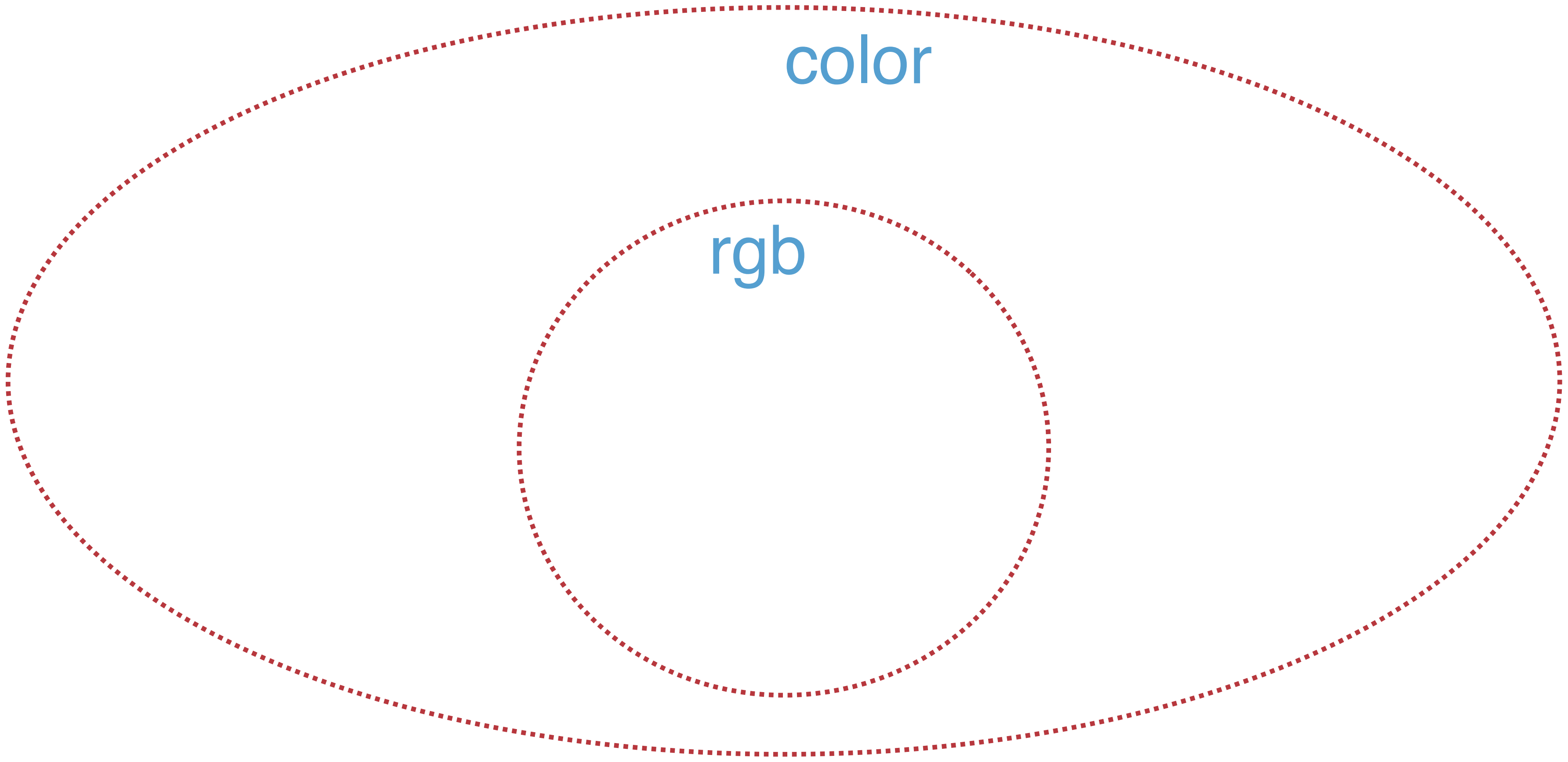
- Define a pretty printer for opening a tag
  - (* pp (h 1) = “<h1>” *) *)
  - Assume we have a natToString function

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

So Far:

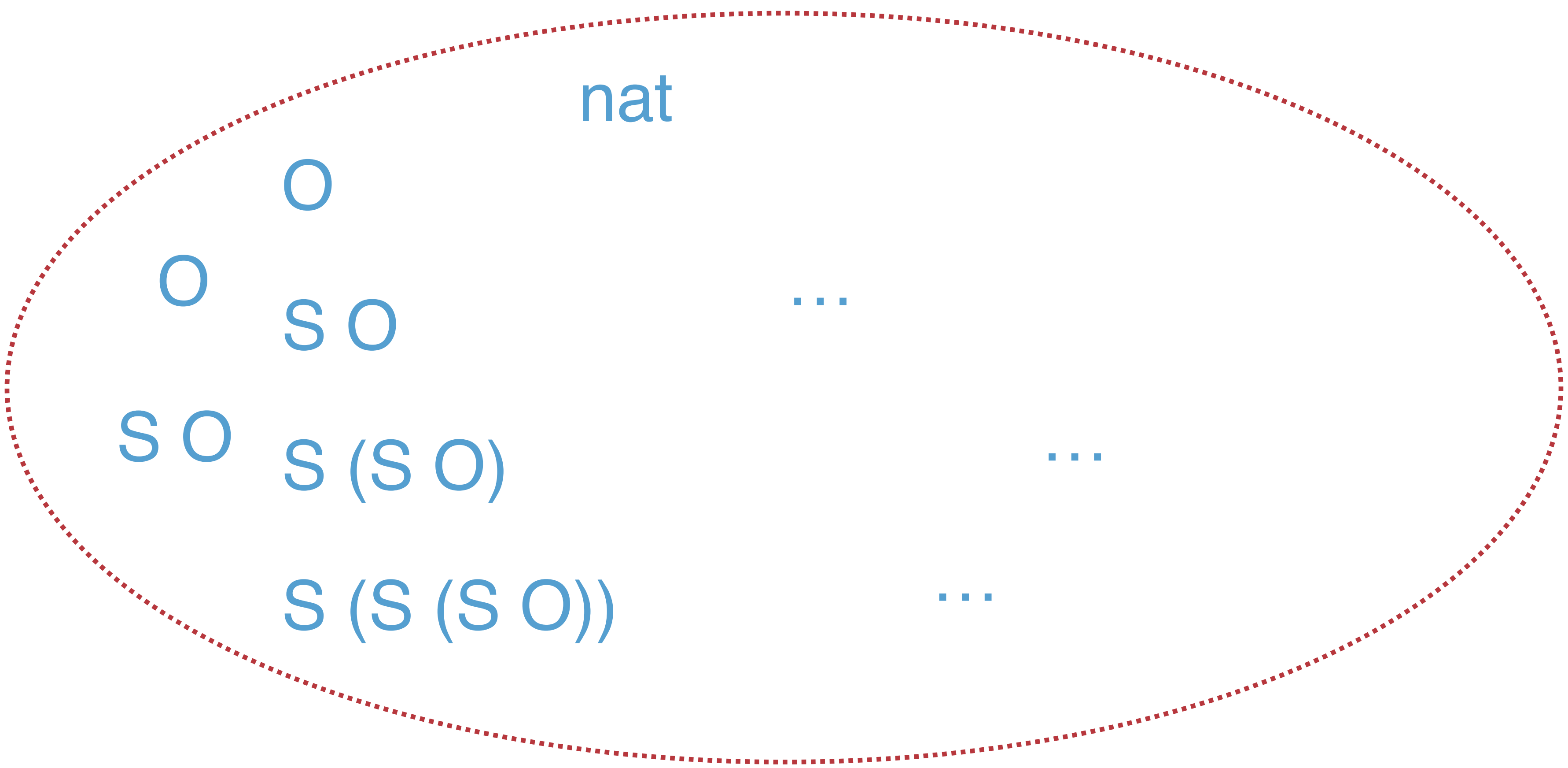

```
Inductive rgb : Type := | red | green | blue.

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 3 = 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**
  - Rocq 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**
  - Rocq 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**
  - ★ Rocq 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)

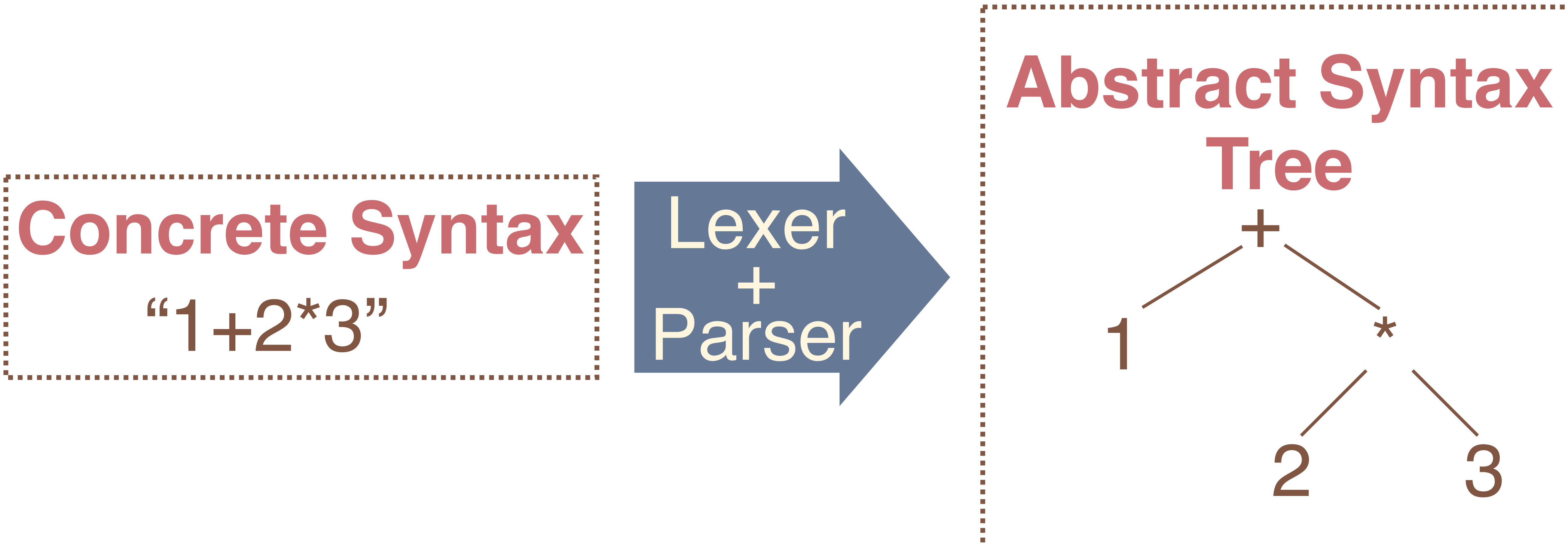
Backus-Naur Form (BNF) Definitions:

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).
```

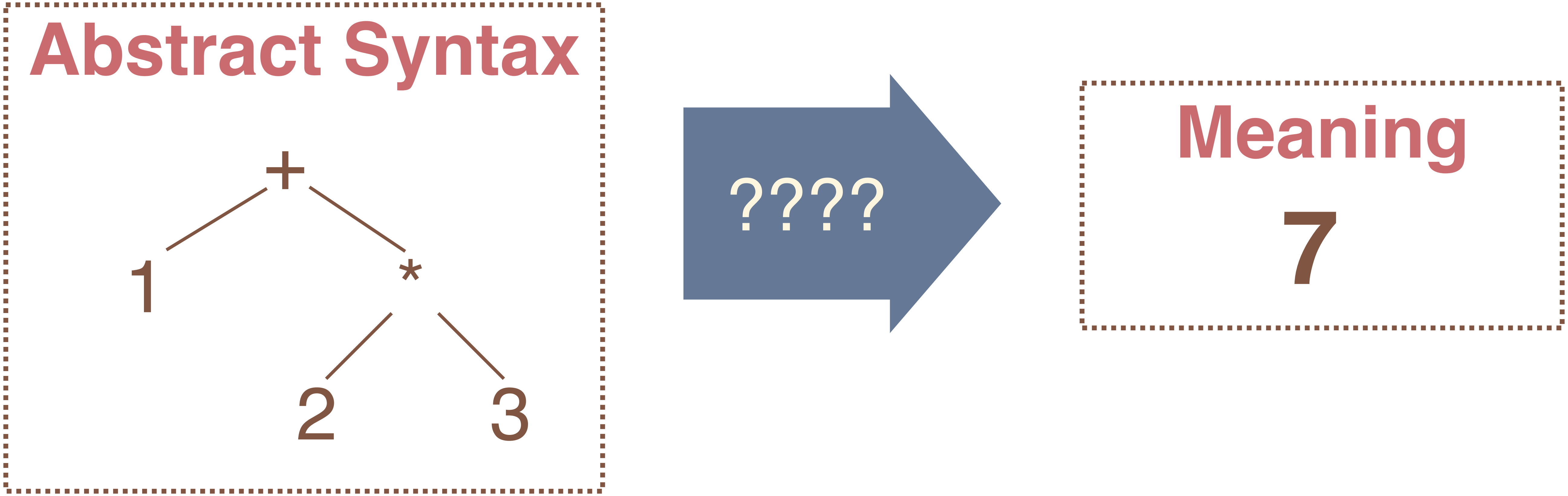
- ★ One constructor per rule
- ★ Nonterminal = inductive type being defined

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.
```