

# **Inference Rules**

**Concepts of Programming Languages**  
**Lecture 4**

# Outline

- » Discuss Formal Typing/Semantic Rules
- » Look at example rules for the constructs we've seen so far
- » Learn to read inference rules, i.e., translate mathematical notation to English and English to mathematical notation

# Recall: Integer Addition (Informal)

$$13 + x$$

# Recall: Integer Addition (Informal)

$$13 + x$$

**Syntax:**  $\text{EXPRESSION}_1 + \text{EXPRESSION}_2$

# Recall: Integer Addition (Informal)

$$13 + x$$

**Syntax:**  $\text{EXPRESSION}_1 + \text{EXPRESSION}_2$

**Typing:**  $\text{EXPRESSION}_1$  *must be an Integer*;  $\text{EXPRESSION}_2$  *must be an Integer*; then  $\text{EXPRESSION}_1 + \text{EXPRESSION}_2$  *will be an integer*

# Recall: Integer Addition (Informal)

13 + x
--------

**Syntax:**  $\text{EXPRESSION}_1 + \text{EXPRESSION}_2$

**Typing:**  $\text{EXPRESSION}_1$  *must be an Integer*;  $\text{EXPRESSION}_2$  *must be an Integer*; then  $\text{EXPRESSION}_1 + \text{EXPRESSION}_2$  *will be an integer*

**Semantics:** *Evaluate*  $\text{EXPRESSION}_1$ ; *say it has value*  $v_1$ ;  
*Evaluate*  $\text{EXPRESSION}_2$ ; *say it has value*  $v_2$ ; *add*  $v_1$  *and*  $v_2$

Let's start with the  
formal syntax

# Note: Production Rules and Syntax

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$



# Note: Production Rules and Syntax

`<expr> ::= <expr> + <expr>`

This is called a *production rule* and is part of a *BNF grammar*

# Note: Production Rules and Syntax

$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$$

This is called a *production rule* and is part of a **BNF grammar**

**This reads as:** if  $e_1$  is a well-formed expression and  $e_2$  is a well-formed expression, then  $e_1 + e_2$  is a well-formed expression

# Note: Production Rules and Syntax

$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$$

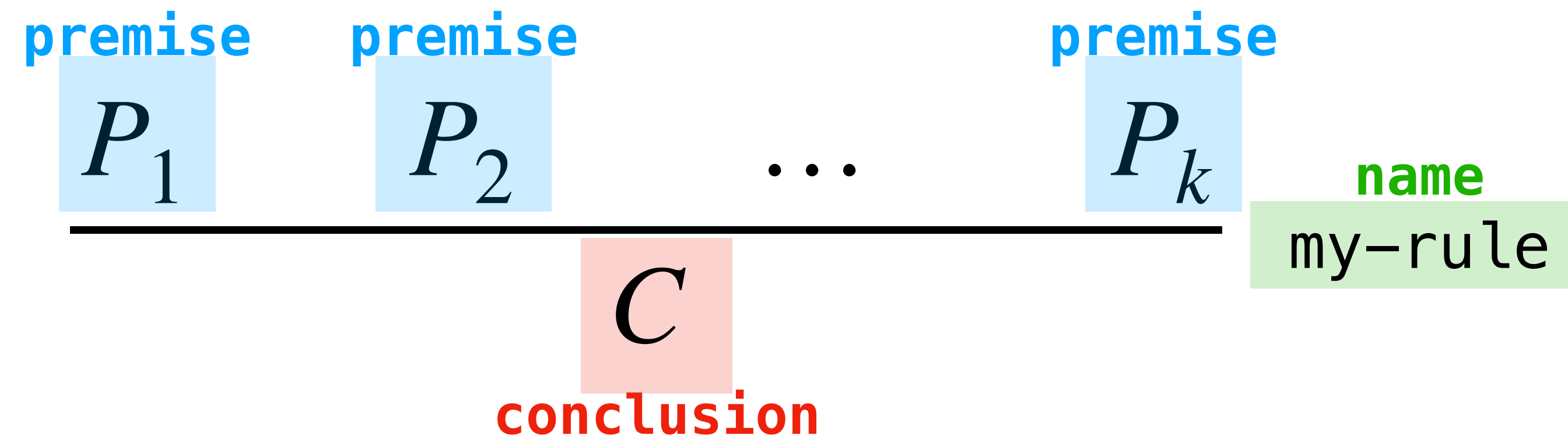
This is called a *production rule* and is part of a *BNF grammar*

**This reads as:** if  $e_1$  is a well-formed expression and  $e_2$  is a well-formed expression, then  $e_1 + e_2$  is a well-formed expression

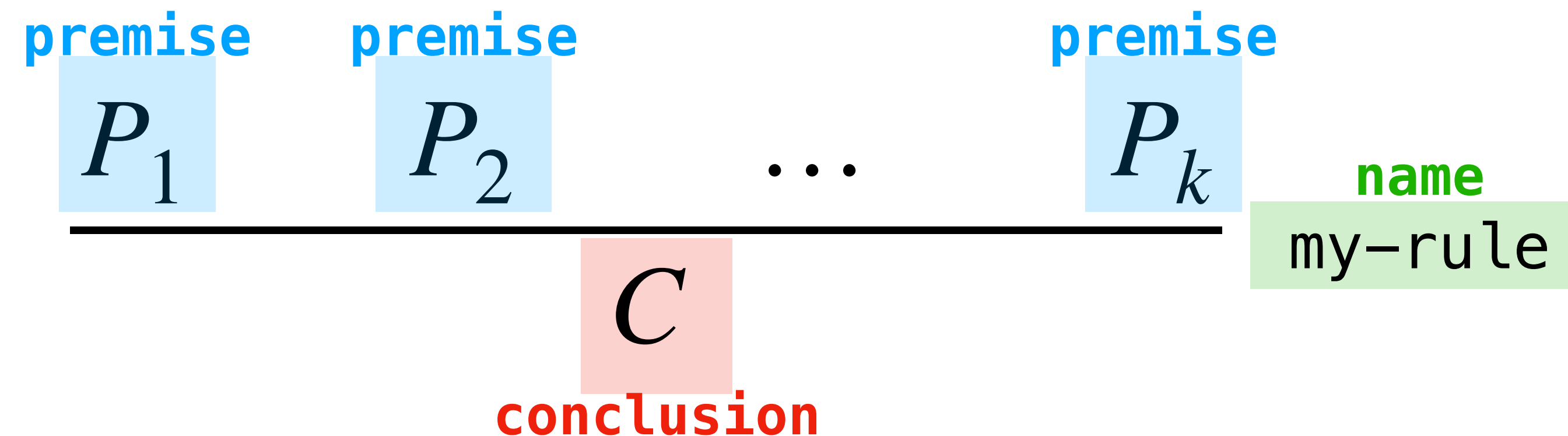
We won't focus on this until the second half of the course but you should start to get comfortable with the syntax

# Inference Rules

# Inference Rules

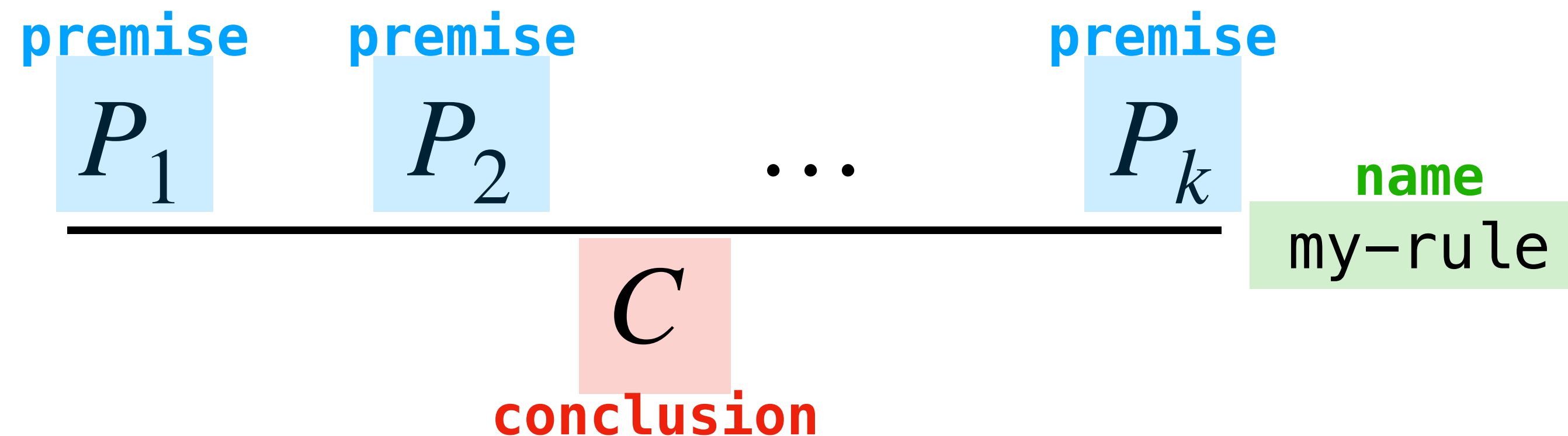


# Inference Rules



Then general form of an inference rule has a collection of **premises** and a **conclusion**

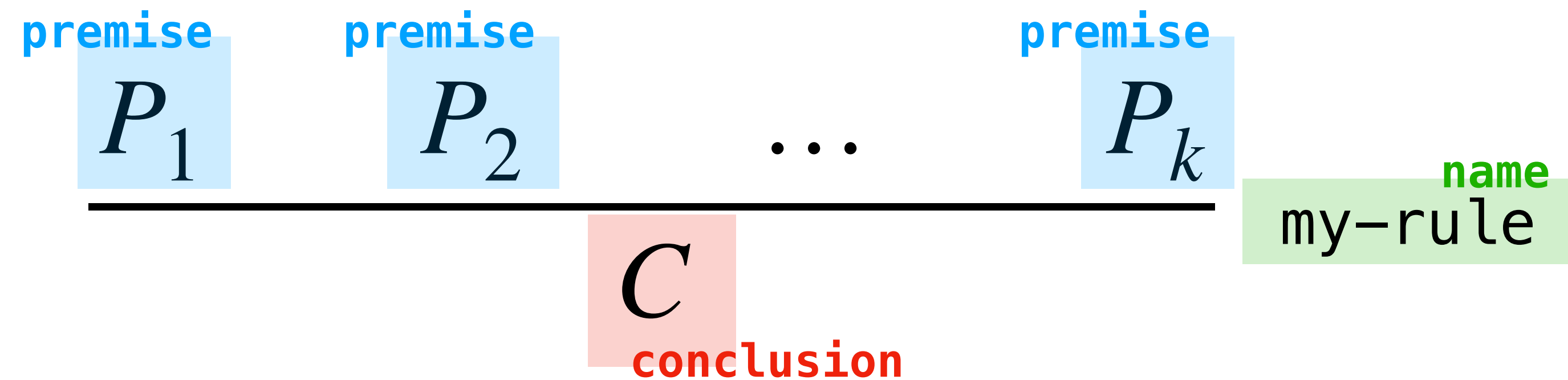
# Inference Rules



Then general form of an inference rule has a collection of **premises** and a **conclusion**

There may be no premises, this is called an **axiom**

# Inference Rules



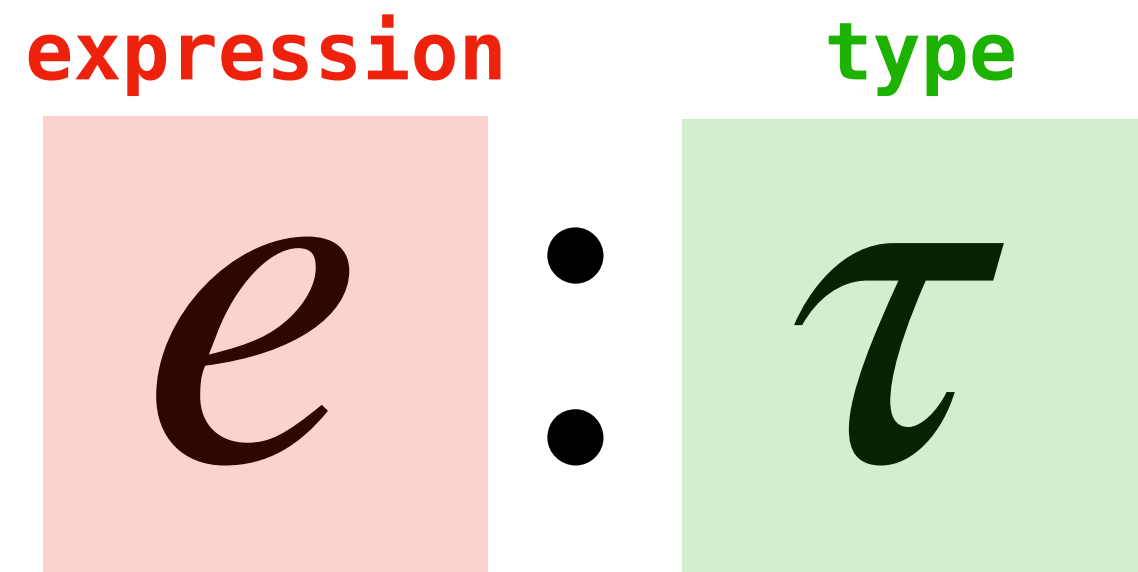
We can read this as:

*If  $P_1$  holds and  $P_2$  holds and ...  $P_k$  holds, then  $C$  holds (by **my-rule**)*



# Typing Judgment and Rules

# Typing Judgment and Rule



A typing judgment is a formal way of representing the statement:

*$e$  has type  $\tau$*

*A typing rule is an inference rule whose premises and conclusion are typing judgments*

# Typing Rule for Addition

We will use inference rules both to express typing rules and semantics rules

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}} \text{ (addInt)}$$

Say it in English!

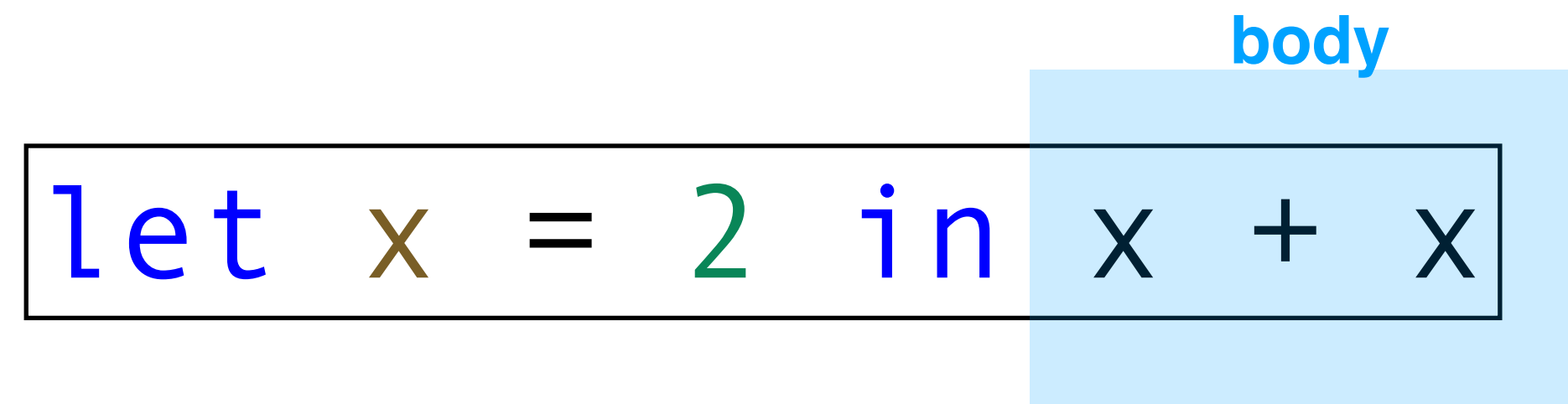
*If ( $e_1$  has type **int**) holds and ( $e_2$  has type **int**) holds, then ( $e_1 + e_2$  has type **int**) holds (by **addInt**)*

But this breaks down as soon  
as we encounter variables

# Let Expressions (Informal)

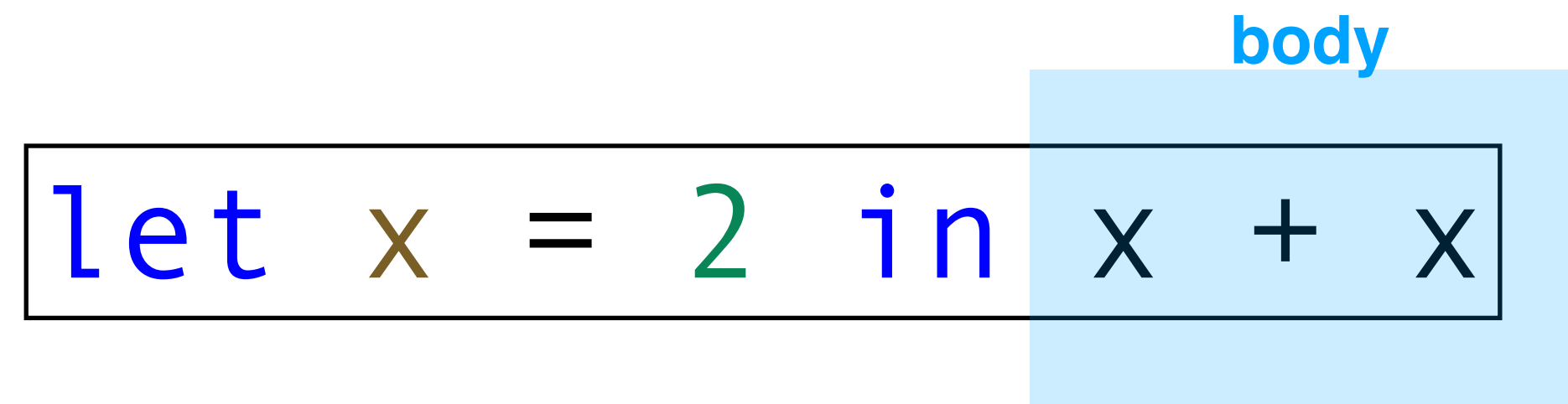
body

```
let x = 2 in x + x
```



The diagram illustrates the structure of a let expression. The text "let x = 2 in x + x" is enclosed in a black rectangular box. The word "let" is blue, "x" is brown, "=" is black, "2" is green, "in" is blue, "x" is black, "+" is black, and the second "x" is black. A light blue rectangular box highlights the right portion of the expression, containing "x + x". The word "body" is written in blue text above the light blue box, indicating that the expression "x + x" is the body of the let expression.

# Let Expressions (Informal)



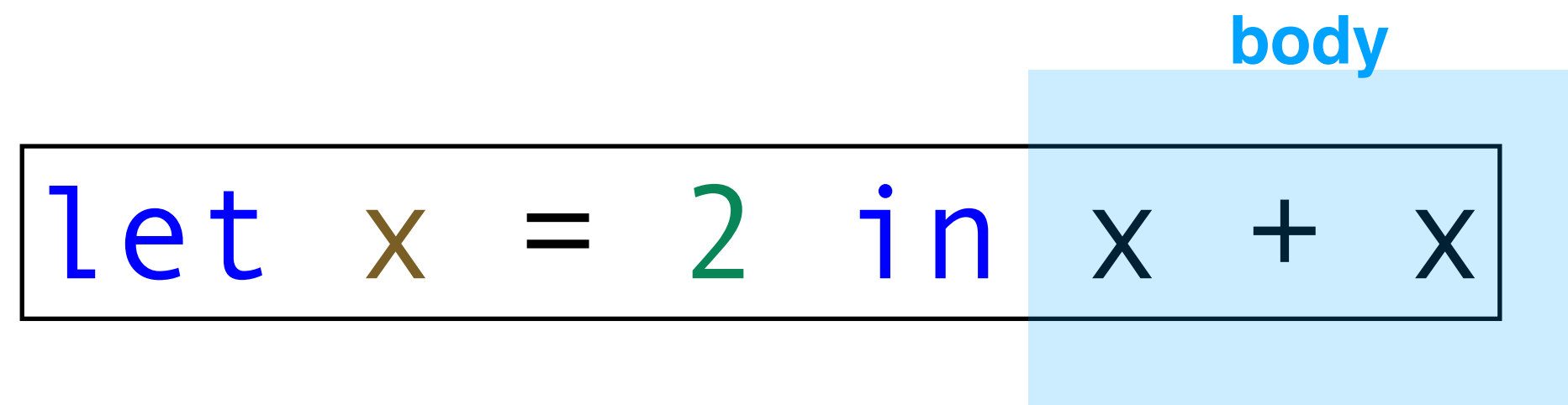
The diagram shows the code `let x = 2 in x + x` with a thin black border. The word `let` is blue, `x` is brown, `=` is black, `2` is green, `in` is blue, and `x + x` is black. A light blue rectangular box highlights the entire expression, and the word `body` in blue is positioned above the right side of this box.

```
let x = 2 in x + x
```

**syntax:** `let VARIABLE = EXPRESSION in BODY`

# Let Expressions (Informal)

body

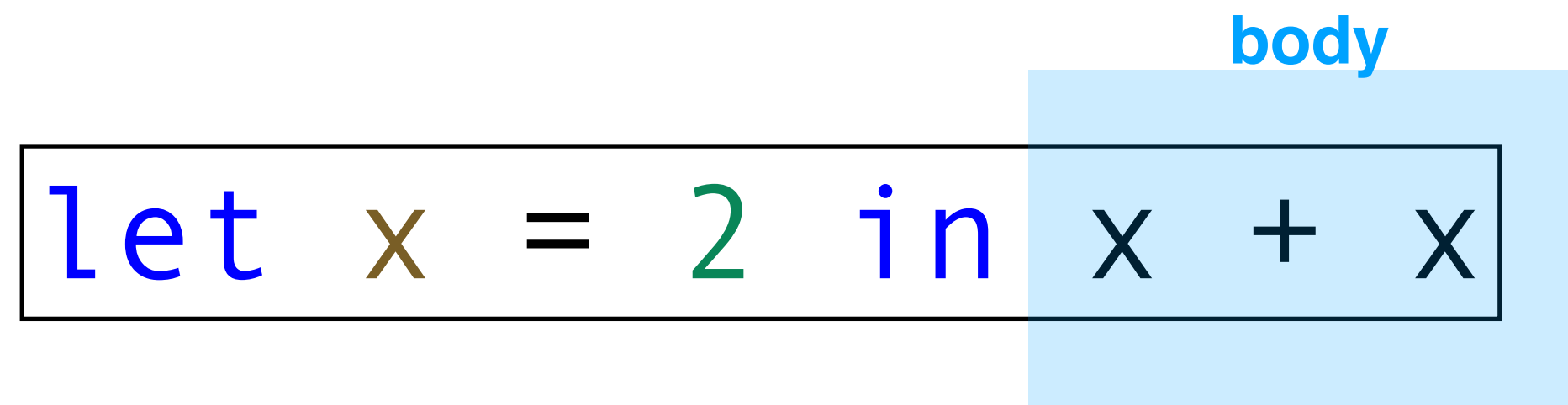


The diagram shows the code `let x = 2 in x + x` with syntax highlighting: `let` is blue, `x` is brown, `=` is black, `2` is green, `in` is blue, and `x + x` is black. A light blue rectangular box labeled "body" in blue text at the top right encloses the expression `x + x`. A thin black rectangular box encloses the entire code snippet.

**syntax:** `let VARIABLE = EXPRESSION in BODY`

**typing:** *compute type of* EXPRESSION; *assume*  
VARIABLE *has that type; compute type of* BODY

# Let Expressions (Informal)



The diagram shows the code `let x = 2 in x + x` with syntax highlighting: `let` is blue, `x` is brown, `=` is green, `2` is green, `in` is blue, `x` is brown, `+` is green, and `x` is brown. A light blue rectangular box highlights the entire expression, and the word `body` in blue is positioned above the right side of this box.

**syntax:** `let VARIABLE = EXPRESSION in BODY`

**typing:** *compute type of* EXPRESSION; *assume*  
VARIABLE *has that type; compute type of* BODY

**semantics:** *compute value of* EXPRESSION; *substitute*  
*that value for* VARIABLE *in* BODY



# Let-Expressions (Syntax Rule)

$\langle \text{expr} \rangle ::= \text{let } \langle \text{var} \rangle = \langle \text{expr} \rangle \text{ in } \langle \text{expr} \rangle$

If  $x$  is a valid variable name, and  $e_1$  is a well-formed expression and  $e_2$  is a well-formed expression then

$\text{let } x = e_1 \text{ in } e_2$

is a well-formed expression

# Let's Try to Write the Typing Rule

We will use inference rules both to express typing rules and semantics rules

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{\text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{ (let)}$$

Say it in English!

*If  $(e_1 \text{ has type } \tau_1)$  holds and  $(e_2 \text{ has type } \tau_2)$  holds, then  $(\text{let } x = e_1 \text{ in } e_2 \text{ has type } \tau_2)$  holds (by **let**)*

# Let's Try to Write the Typing Rule

We will use inference rules both to express typing rules and semantics rules

assuming  $x$  has type  $\tau_1$ ?

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{\text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{ (let)}$$

Say it in English!

*If  $(e_1 \text{ has type } \tau_1)$  holds and  $(e_2 \text{ has type } \tau_2)$  holds, then  $(\text{let } x = e_1 \text{ in } e_2 \text{ has type } \tau_2)$  holds (by **let**)*

# Example

`let x = 2 in x`

$$\frac{2 : \text{int} \quad x : \tau_2}{\text{let } x = 2 \text{ in } x : \tau_2} \text{ (let)}$$

# Example

`let x = 2 in x`

What is  $\tau_2$ ?

$$\frac{2 : \text{int} \quad x : \tau_2}{\text{let } x = 2 \text{ in } x : \tau_2} \text{ (let)}$$

# Example

`let x = 2 in x`

$$\frac{2 : \text{int} \quad x : \tau_2 \text{ What is } \tau_2?}{\text{let } x = 2 \text{ in } x : \tau_2} \text{ (let)}$$

We need to carry over the information that  $x$  has type `int` to the second premise. How do we do that?

# Contexts

$$\Gamma = \{ x : \text{int}, y : \text{string}, z : \text{int} \rightarrow \text{string} \}$$

# Contexts

$$\Gamma = \{ x : \text{int}, y : \text{string}, z : \text{int} \rightarrow \text{string} \}$$

A context is a set of variable declarations



# Contexts

$$\Gamma = \{ x : \text{int}, y : \text{string}, z : \text{int} \rightarrow \text{string} \}$$

A **context** is a set of **variable declarations**

A variable declaration  $(x : \tau)$  says: "I declare that the variable  $x$  is of type  $\tau$ "

# Contexts

$$\Gamma = \{ x : \text{int}, y : \text{string}, z : \text{int} \rightarrow \text{string} \}$$

A **context** is a set of **variable declarations**

A variable declaration  $(x : \tau)$  says: "I declare that the variable  $x$  is of type  $\tau$ "

A context keeps track of all the types of variables in the "(static) environment" (or scope)

# Typing Judgments

$$\begin{array}{c} \text{context} \\ \Gamma \end{array} \vdash \begin{array}{c} \text{expression} \\ e \end{array} : \begin{array}{c} \text{type} \\ \tau \end{array}$$

A typing judgment is a compact way of representing the statement:

*$e$  is of type  $\tau$  in the context  $\Gamma$*

A **typing rule** is an inference rule whose premises and conclusion are typing judgments

# Example: Using Typing Judgements

$$\{x : \text{int}\} \vdash x : \text{int}$$

# Example: Using Typing Judgements

$$\{x : \text{int}\} \vdash x : \text{int}$$

**In English:** *Given I declare that  $x$  is an  $\text{int}$ ,  
the expression  $x$  is an  $\text{int}$*

# Example: Using Typing Judgements

$$\{x : \text{int}\} \vdash x : \text{int}$$

**In English:** *Given I declare that  $x$  is an  $\text{int}$ , the expression  $x$  is an  $\text{int}$*

The context allows us to determine the type of an expression *relative to the types of variables*

# Example: Using Typing Judgements

$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$

# Example: Using Typing Judgements

$$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$$

**In English:** *Given I declare that  $b$  has type  $\text{bool}$ , the expression  $\text{if } b \text{ then } 2 \text{ else } 3$  has type  $\text{int}$*



# Example: Using Typing Judgements

$$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$$

**In English:** *Given I declare that  $b$  has type  $\text{bool}$ , the expression  $\text{if } b \text{ then } 2 \text{ else } 3$  has type  $\text{int}$*

The context allows us to determine the type of an expression *relative to the types of variables*

# Integer Addition Typing Rule

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ (addInt)}$$

*If ( $e_1$  has type **int** in any context  $\Gamma$ ) holds and ( $e_2$  has type **int** in the same context  $\Gamma$  holds, then ( $e_1 + e_2$  has type **int**) holds (by **addInt**)*

# Let-Expressions (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \text{ (let)}$$

If  $e_1$  is of type  $\tau_1$  in the context  $\Gamma$ , and  $e_2$  is of type  $\tau$  in the context  $\Gamma$  *with the variable declaration*  $(x : \tau_1)$  *added to it*, then

$\text{let } x = e_1 \text{ in } e_2$

is of type  $\tau$  in the context  $\Gamma$

# Let-Expressions (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \text{ (let)}$$

**Note: Look at how much more compact the rule is!**

If  $e_1$  is of type  $\tau_1$  in the context  $\Gamma$ , and  $e_2$  is of type  $\tau$  in the context  $\Gamma$  *with the variable declaration*  $(x : \tau_1)$  *added to it*, then

$\text{let } x = e_1 \text{ in } e_2$

is of type  $\tau$  in the context  $\Gamma$

# If-Expressions (Informal)

```
if x > 0 then x else -x
```

# If-Expressions (Informal)

```
if x > 0 then x else -x
```

**Syntax:** if CONDITION then TRUE-CASE else FALSE-CASE

# If-Expressions (Informal)

```
if x > 0 then x else -x
```

**Syntax:** if CONDITION then TRUE-CASE else FALSE-CASE

**Typing:** CONDITION *must be a Boolean; compute the types of*  
TRUE-CASE *and* FALSE-CASE; *must be the same type; expression*  
*type is same as that of* TRUE-CASE *and* FALSE-CASE

# If-Expressions (Informal)

```
if x > 0 then x else -x
```

**Syntax:** if CONDITION then TRUE-CASE else FALSE-CASE

**Typing:** CONDITION *must be a Boolean; compute the types of* TRUE-CASE *and* FALSE-CASE; *must be the same type; expression type is same as that of* TRUE-CASE *and* FALSE-CASE

**Semantics:** *If* CONDITION *evaluates to true; evaluate* TRUE-CASE, *else evaluate* FALSE-CASE



# If-Expressions (Informal)

```
if x > 0 then x else -x
```

**Syntax:** if CONDITION then TRUE-CASE else FALSE-CASE

**Typing:** CONDITION *must be a Boolean; compute the types of* TRUE-CASE *and* FALSE-CASE; *must be the same type; expression type is same as that of* TRUE-CASE *and* FALSE-CASE

**Semantics:** *If* CONDITION *evaluates to true; evaluate* TRUE-CASE, *else evaluate* FALSE-CASE

**Formal Syntax:**  $\langle \text{expr} \rangle ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle$

# If-Expressions (Syntax Rule)

$\langle \text{expr} \rangle ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle$

If  $e_1$  is a well-formed expression and  $e_2$  is a well-formed expression and  $e_3$  is a well-formed expression, then

$\text{if } e_1 \text{ then } e_2 \text{ else } e_3$

is a well-formed expression

# If-Expressions (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{(if)}$$

If  $e_1$  is of type `bool` in the context  $\Gamma$  and  $e_2$  and  $e_3$  are of type  $\tau$  in the context  $\Gamma$ , then

`if  $e_1$  then  $e_2$  else  $e_3$`

is of type  $\tau$  in the context  $\Gamma$

# Note: Judgements are claims

$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{string}$

# Note: Judgements are claims

`{b : bool} ⊢ if b then 2 else 3 : string`

A judgement is a *claim* in the same way that "there are infinitely many twin primes" or "pigs fly" is a claim

# Note: Judgements are claims

$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{string}$

A judgement is a *claim* in the same way that "there are infinitely many twin primes" or "pigs fly" is a claim

We haven't **proved** anything by writing down a typing judgment

# Note: Judgements are claims

$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{string}$

A judgement is a *claim* in the same way that "there are infinitely many twin primes" or "pigs fly" is a claim

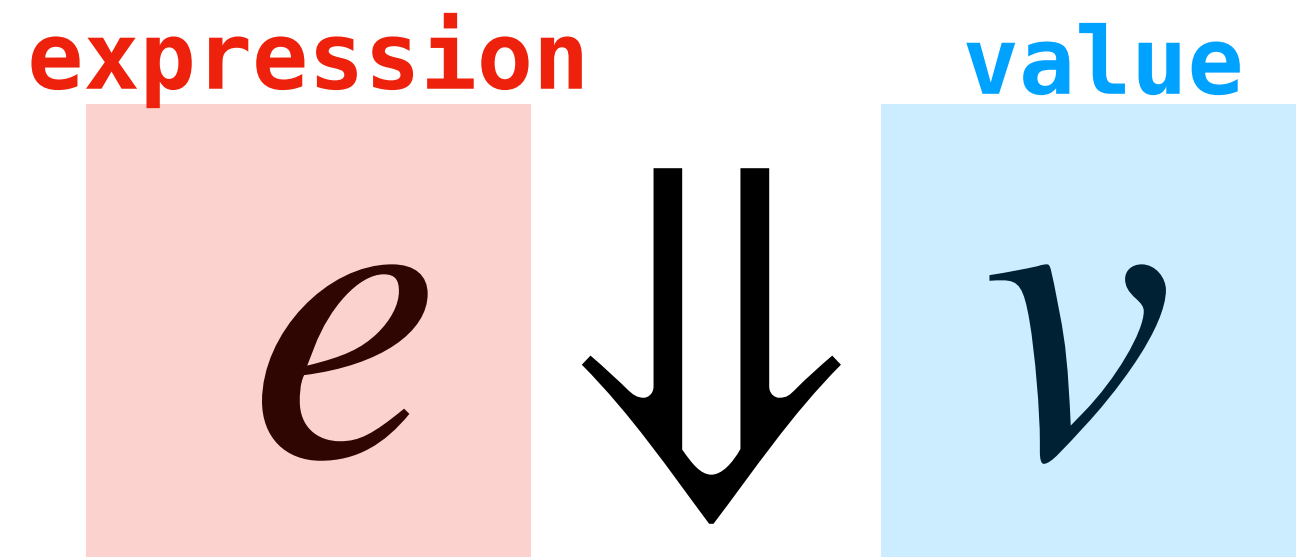
We haven't **proved** anything by writing down a typing judgment

**Next week:** We will talk about **typing derivations**, which are used to demonstrate that expressions *actually* have their desired types in our PL

# **Semantic Judgment and Rules**



# Semantic Judgements



A semantic judgement is a compact way of representing the statement:

*The expression  $e$  evaluates to the value  $v$*

A **semantic rule** is an inference rule with semantic judgements

# Example: Reading Semantic Judgments

if 2 > 3 then 2 + 2 else 3  $\Downarrow$  3

**In English:** The expression

if 2 > 3 then 2 + 2 else 3

evaluates to the value 3

# Recall: Integer Addition Semantic Rule

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 + v_2} \text{ (evalInt)}$$

*If  $e_1$  evaluates to the (integer)  $v_1$  and  $e_2$  evaluates to the (integer)  $v_2$ , then  $e_1 + e_2$  evaluates to the (integer)  $v_1 + v_2$*

# Let-Expressions (Semantic Rule)

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\text{let } x = e_1 \text{ in } e_2 \Downarrow v} \text{ (letEval)}$$

If  $e_1$  evaluates to  $v_1$  and  $e_2$  with  $v_1$  substituted for  $x$  evaluates to  $v$ , then

$\text{let } x = e_1 \text{ in } e_2$

evaluates to  $v$

# Note: Values are not Expressions

if 2 > 3 then 2 + 2 else 3  $\Downarrow$  3

We will draw a distinction between values and expressions (note the font and color difference)

**Example.** We'll use regular numbers to represent integer values, and we'll use  $\top$  and  $\perp$  for the true and false Boolean values

# If-Expressions (Semantic Rule 1)

$$\frac{e_1 \Downarrow T \quad e_2 \Downarrow v_2}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_2} \text{ (ifEvalTrue)}$$

If  $e_1$  evaluates to  $T$  and  $e_2$  evaluates to  $v_2$ , then

$\text{if } e_1 \text{ then } e_2 \text{ else } e_3$

evaluates to  $v_2$

# If-Expressions (Semantic Rule 2)

$$\frac{e_1 \Downarrow \perp \quad e_3 \Downarrow v_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_3} \text{ (ifEvalFalse)}$$

If  $e_1$  evaluates to  $\perp$  and  $e_2$  evaluates to  $v_2$ , then

$\text{if } e_1 \text{ then } e_2 \text{ else } e_3$

evaluates to  $v_3$

# If-Expressions (Semantic Rule 2)

$$\frac{e_1 \Downarrow \perp \quad e_3 \Downarrow v_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_3} \text{ (ifEvalFalse)}$$

**Note: we never evaluate both branches**

If  $e_1$  evaluates to  $\perp$  and  $e_2$  evaluates to  $v_2$ , then

**if**  $e_1$  **then**  $e_2$  **else**  $e_3$

evaluates to  $v_3$



Questions?

# **Understanding Check: Functions and Applications**

# Functions (Syntax Rule)

$\langle \text{expr} \rangle ::= \text{fun } \langle \text{var} \rangle \rightarrow \langle \text{expr} \rangle$

# Functions (Syntax Rule)

$$\langle \text{expr} \rangle ::= \text{fun } \langle \text{var} \rangle \rightarrow \langle \text{expr} \rangle$$

If  $x$  is a valid variable name and  $e$  is a well-formed expression, then

$$\text{fun } x \rightarrow e$$

is a well-formed expression

# Functions (Typing Rule)

$$\frac{\rho, x : \tau_1 \vdash e : \tau_2}{\rho \vdash \boxed{\text{fun } x \rightarrow e} : \tau_1 \rightarrow \tau_2} \text{(fun)}$$

# Functions (Typing Rule)

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2} \text{ (fun)}$$

If  $e$  has type  $\tau_2$  in the context  $\Gamma$  with the variable declaration  $(x : \tau_1)$  added, then

$\text{fun } x \rightarrow e$

is of type  $\tau_1 \rightarrow \tau_2$  in the context  $\Gamma$

# Functions (Semantic Rule)

---

$$\text{fun } x \rightarrow e \Downarrow (\lambda x. e)$$

(funVal)

---

$$2 \Downarrow 2$$

# Functions (Semantic Rule)

$$\frac{}{\text{fun } x \rightarrow e \Downarrow \lambda x . e} \text{ (funEval)}$$

Under no premises, the expression

$$\text{fun } x \rightarrow e$$

evaluates to the *function value*  $\lambda x . e$  (we'll talk more about function values later)



# Application (Syntax Rule)

$f \ e$

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle \ \langle \text{expr} \rangle$

# Application (Syntax Rule)

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle \langle \text{expr} \rangle$

If  $e_1$  is a well-formed expression and  $e_2$  is a well-formed expression, then  $e_1 e_2$  is a well-formed expression

# Application (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{ (app)}$$

# Application (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \text{ (app)}$$

If  $e_1$  has type  $\tau_2 \rightarrow \tau$  under the context  $\Gamma$  and  $e_2$  has type  $\tau_2$  under the context  $\Gamma$ , then  $e_1 e_2$  has type  $\tau$  under the context  $\Gamma$

# Application (Semantic Rule)

$$\frac{e_1 \Downarrow (\lambda x. e) \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\boxed{e_1 \quad e_2} \Downarrow v}$$

# Application (Semantic Rule)

$$\frac{e_1 \Downarrow \lambda x . e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v} \text{(appEval)}$$

1.  $e_1$  evaluates to a function value  $\lambda x . e$
2.  $e_2$  evaluates to  $v_2$
3.  $e$  with  $v_2$  substituted for  $x$  evaluates to  $v$

It follows that  $e_1 e_2$  evaluates to  $v$

# Homework

Offline, go back to the recap slides at the beginning and compare the formal and informal descriptions...

We'll give a written  
reference for the rules we  
talk about in class

You never need to remember  
any rules in class



# Summary

**Inference rules** formally describe how the typing and semantics of a programming language work

**Tuples** and **records** allow us to group data

**Variants** allow us to organize data by *possible outcomes*