

Inference Rules

Concepts of Programming Languages
Lecture 5

Practice Problem

Determine an expression with the following type

$('a \rightarrow 'a \rightarrow 'b) \rightarrow ('c \rightarrow 'a) \rightarrow 'c \rightarrow 'b$

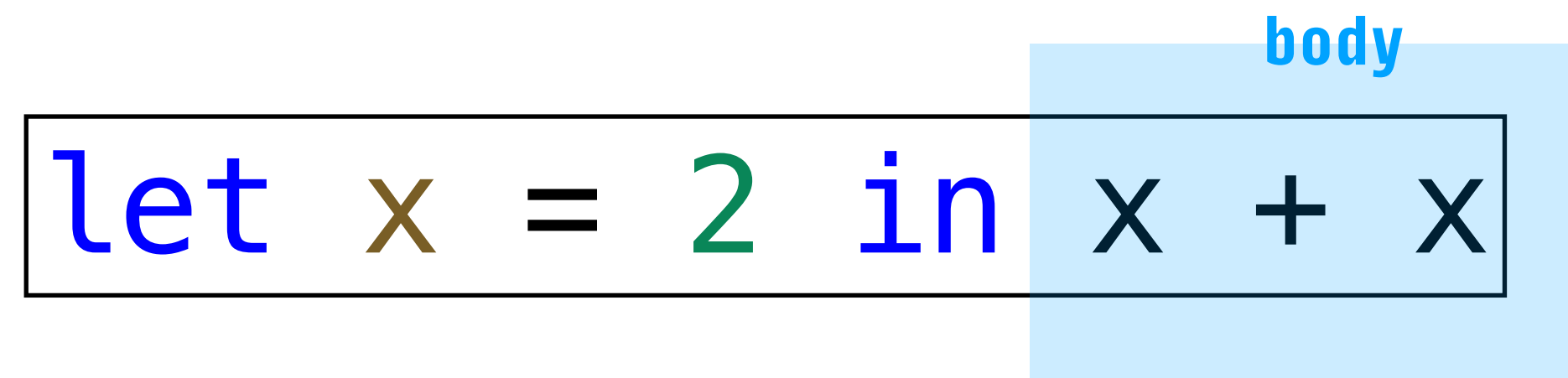
Outline

- » 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

Recap

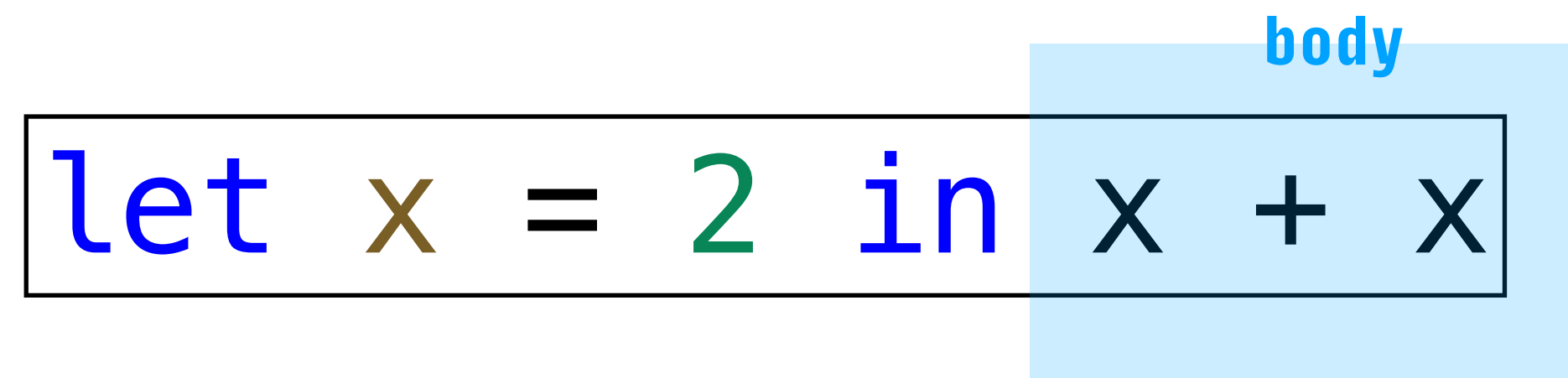
Recall: Local Variables (Informal)

`let x = 2 in x + x`



The diagram illustrates the structure of a `let` expression. The text `let x = 2 in x + x` is shown. The `let` and `in` keywords are blue, `x` is brown, `=` is black, `2` is green, and `x + x` is black. A light blue rectangular box highlights the expression `x + x`, which is the body of the `let` binding. The word `body` is written in blue above the box.

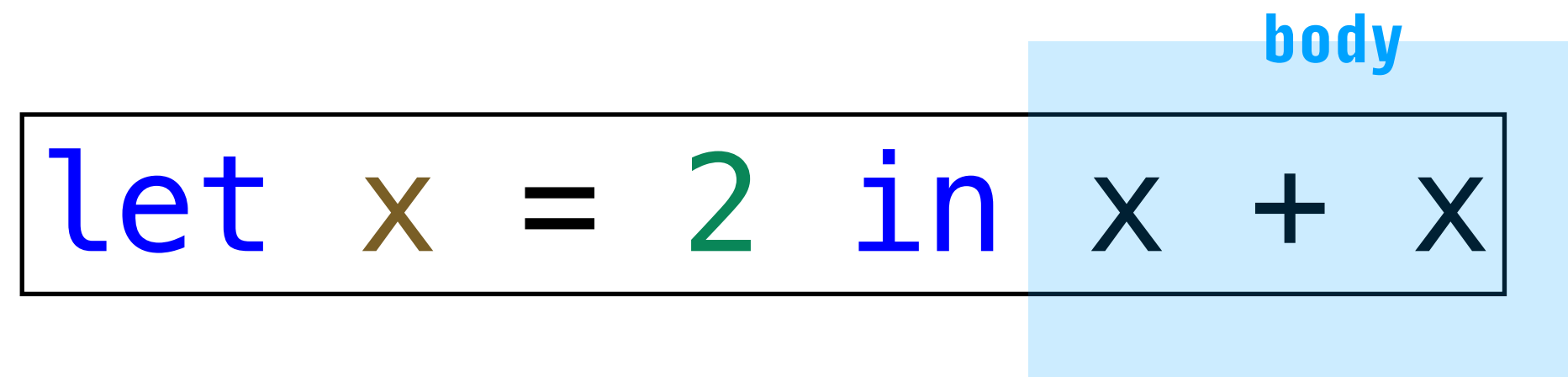
Recall: Local Variables (Informal)



The diagram shows a code snippet `let x = 2 in x + x` enclosed in a black rectangular 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 `x + x` portion. The word `body` is written in blue above the right side of the light blue box.

Syntax: `let VARIABLE = EXPRESSION in BODY`

Recall: Local Variables (Informal)

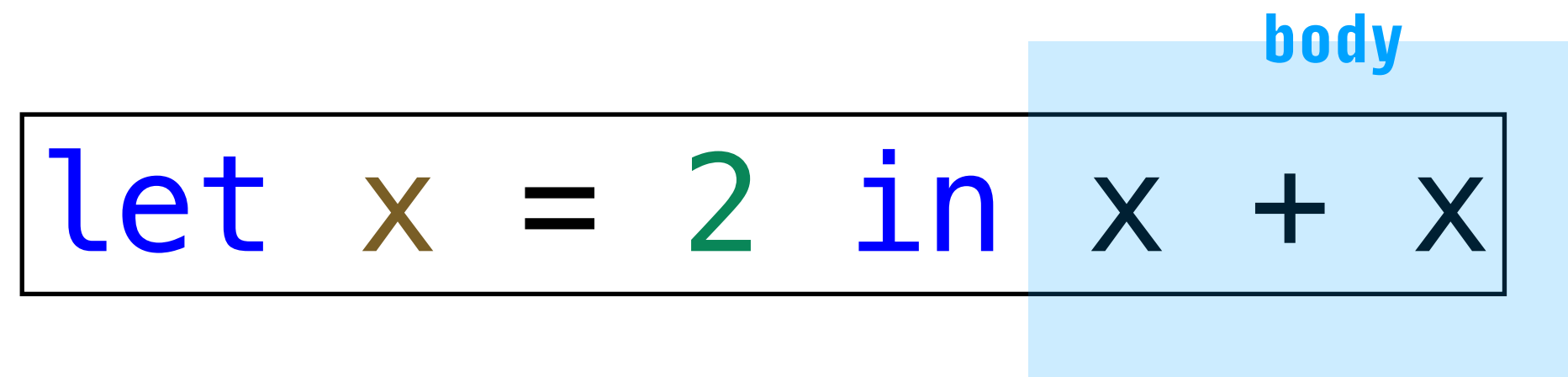


The diagram shows the code `let x = 2 in x + x` enclosed in a black rectangular box. The word `let` is blue, `x` is brown, `=` is black, `2` is green, `in` is blue, and `x + x` is black. A light blue rectangular background highlights the entire expression. The word `body` in blue is positioned above the right side of the box, specifically over the `x + x` part.

Syntax: `let VARIABLE = EXPRESSION in BODY`

Typing: the type is the same as that of `BODY` *given `BODY` is well-typed after substituting the `VARIABLE` in `BODY`*

Recall: Local Variables (Informal)



The diagram shows the code `let x = 2 in x + x` enclosed in a black rectangular box. A light blue rectangular box highlights the expression `x + x` on the right side of the `in` keyword. The word `body` is written in blue text above the right edge of the light blue box.

Syntax: `let VARIABLE = EXPRESSION in BODY`

Typing: the type is the same as that of BODY *given BODY is well-typed after substituting the VARIABLE in BODY*

Semantics: the is the same as the value of BODY *after substituting the VARIABLE in BODY*

let $x = 2$ in $(x + x) : \text{int}$

$: \text{int}$

$: \text{int}$

let $x = 2$ in $x + x \Downarrow 4$

$2 + 2 \Downarrow 4$

Recall: If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Recall: If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

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

Recall: If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

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

Typing: CONDITION must be a Boolean and TRUE-CASE and FALSE-CASE must be the same type. The type is then the same as that of TRUE-CASE and FALSE-CASE

Recall: If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

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

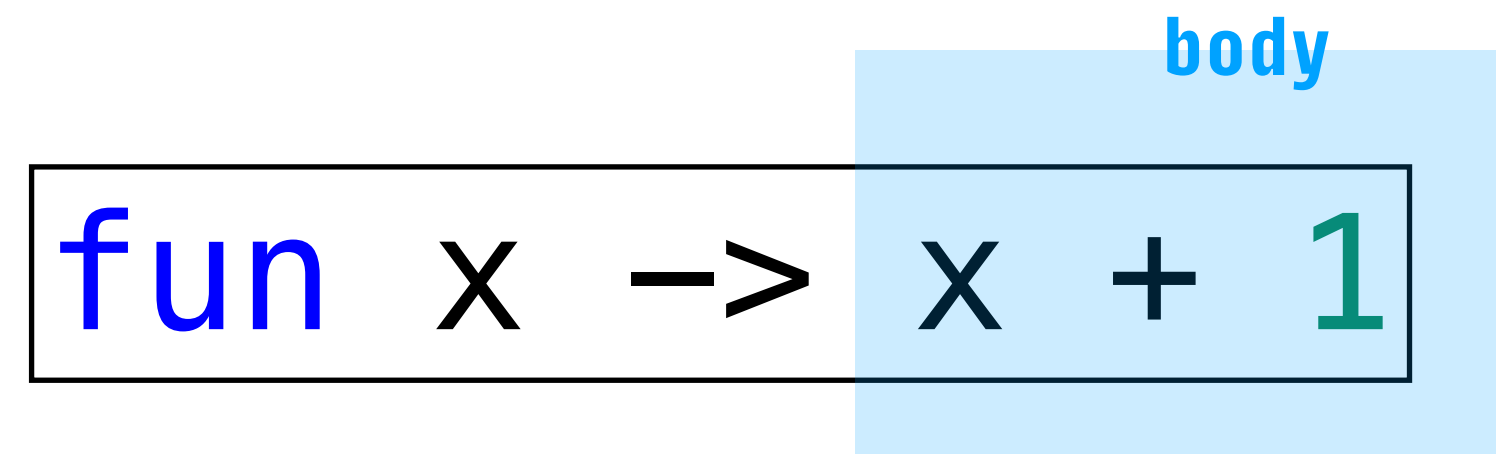
Typing: CONDITION must be a Boolean and TRUE-CASE and FALSE-CASE must be the same type. The type is then the same as that of TRUE-CASE and FALSE-CASE

Semantics: If CONDITION holds, then we get the TRUE-CASE, otherwise we get the FALSE-CASE

Recall: Functions (Informal)

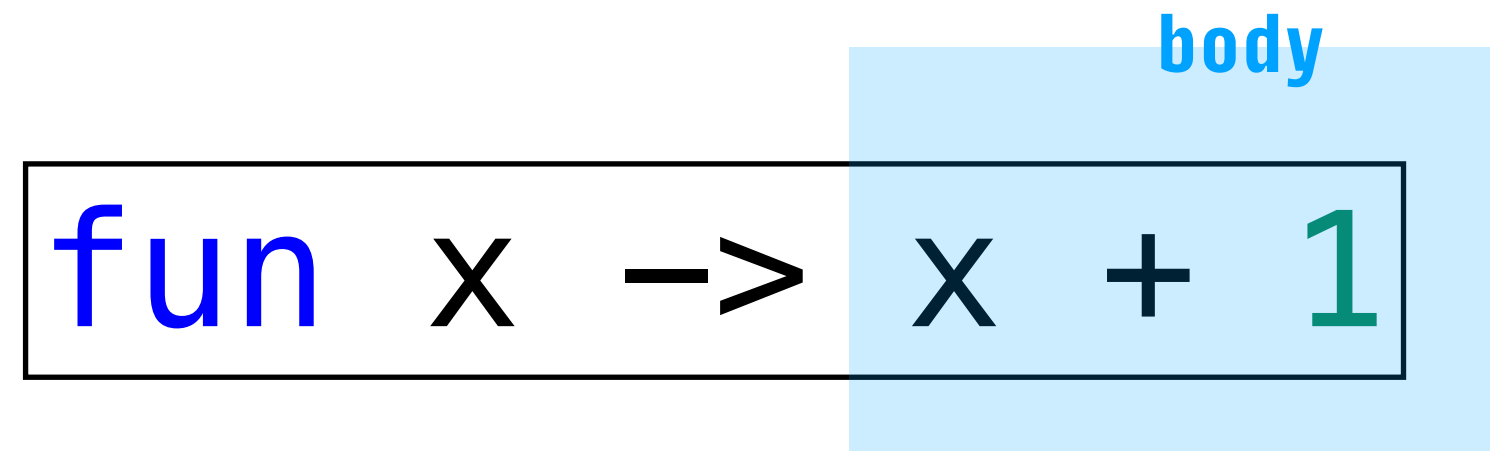
`fun x -> x + 1`

body



The diagram shows the code `fun x -> x + 1` enclosed in a black rectangular box. The text `fun x ->` is in blue, while `x + 1` is in green. A light blue rectangular area highlights the green portion of the code. The word `body` is written in blue above the right side of the light blue area.

Recall: Functions (Informal)

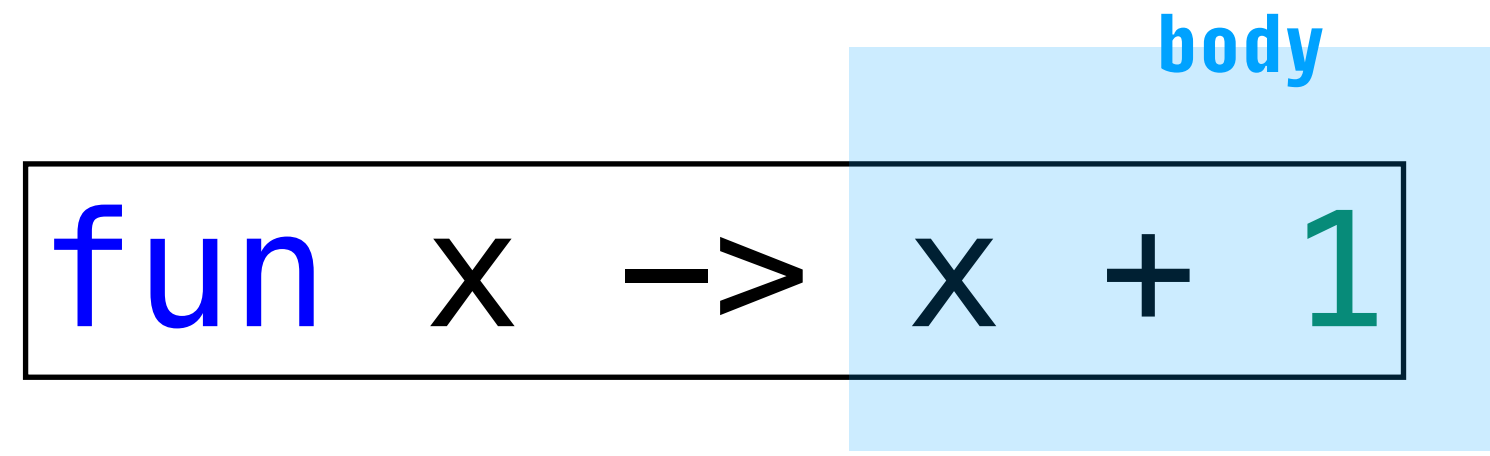


body

```
fun x -> x + 1
```

Syntax: fun VAR-NAME -> EXPR

Recall: Functions (Informal)



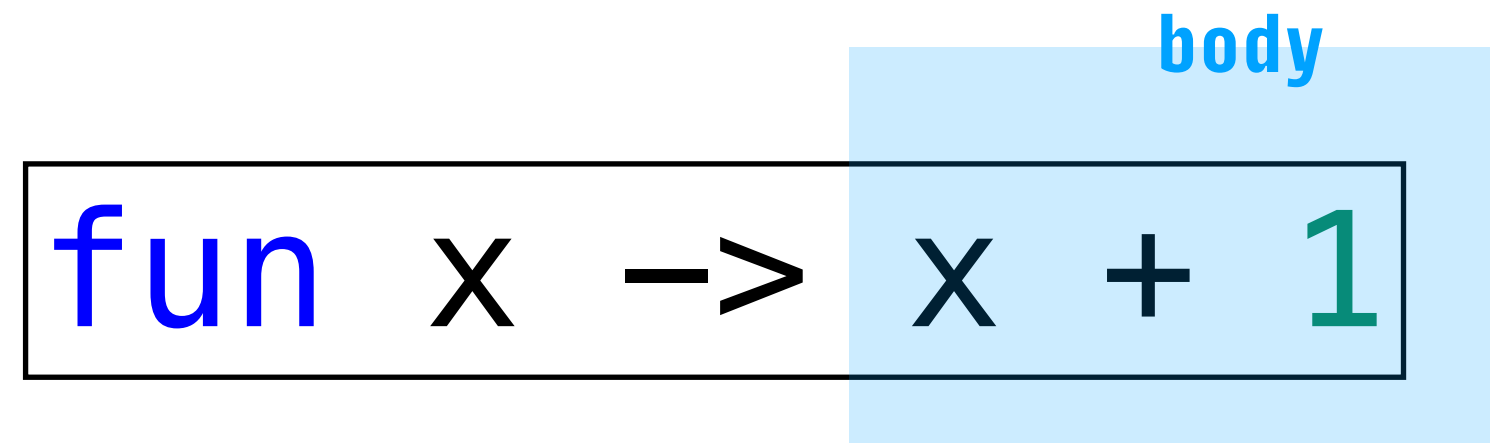
body

```
fun x -> x + 1
```

Syntax: `fun VAR-NAME -> EXPR`

Typing: the type of a function is $T1 \rightarrow T2$ where $T1$ is the type of the input and $T2$ is the type of the output

Recall: Functions (Informal)



body

```
fun x -> x + 1
```

Syntax: `fun VAR-NAME -> EXPR`

Typing: the type of a function is $T1 \rightarrow T2$ where $T1$ is the type of the input and $T2$ is the type of the output

Semantics: A function will evaluate to special *function value* (printed as `<fun>` by utop)

Recall: Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Recall: Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Recall: Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Typing: If FUNCTION-EXPR is of type $T1 \rightarrow T2$,
and ARG-EXPR is of type $T1$, then the type is $T2$

Recall: Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Typing: If FUNCTION-EXPR is of type $T1 \rightarrow T2$,
and ARG-EXPR is of type $T1$, then the type is $T2$

Semantics: Substitute the value of ARG-EXPR into
the body of FUNCTION-EXPR and evaluate that

$(\text{fun } x \rightarrow x + x) \text{ } 2 : \text{int}$

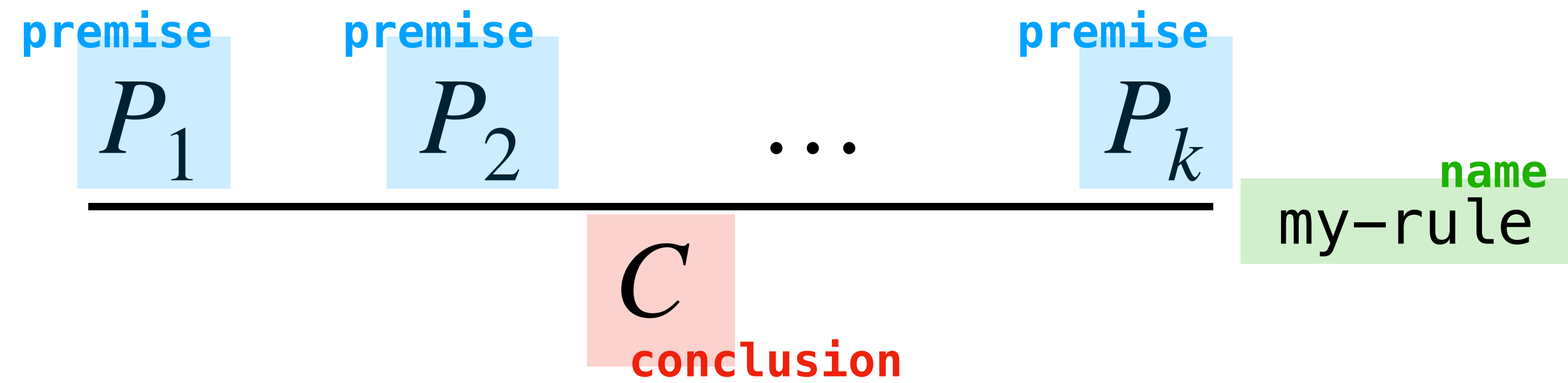
int int
int

$(\text{fun } x \rightarrow x + x) \text{ } 2 \Downarrow 4$

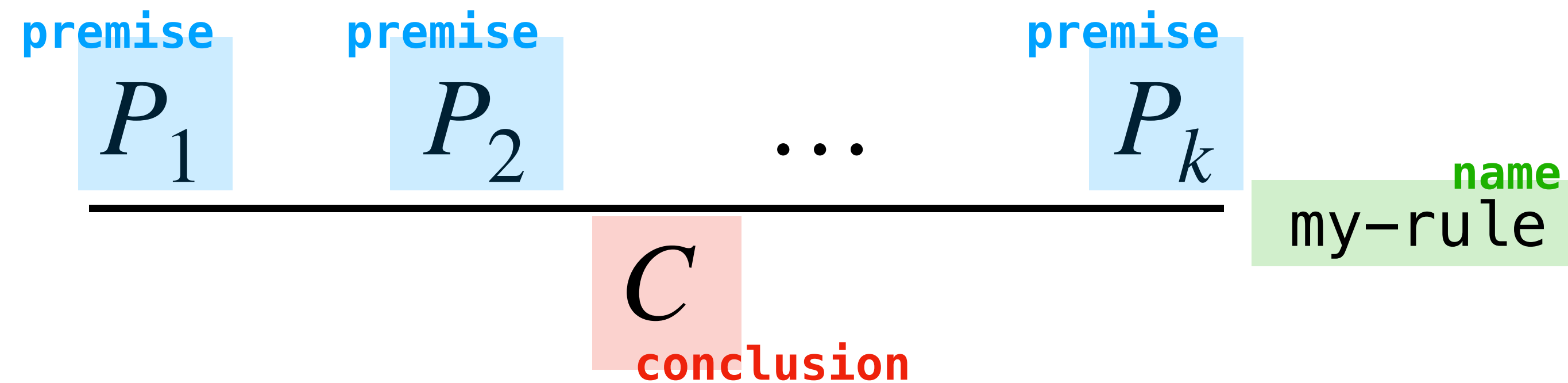
$2 + 2 \Downarrow 4$

Inference Rules

Inference Rules

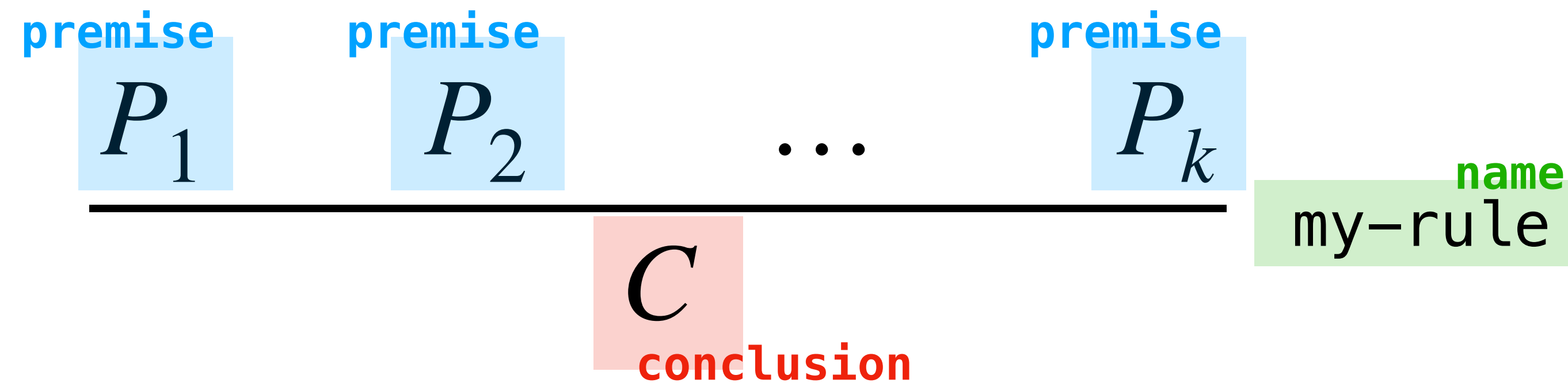


Inference Rules



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

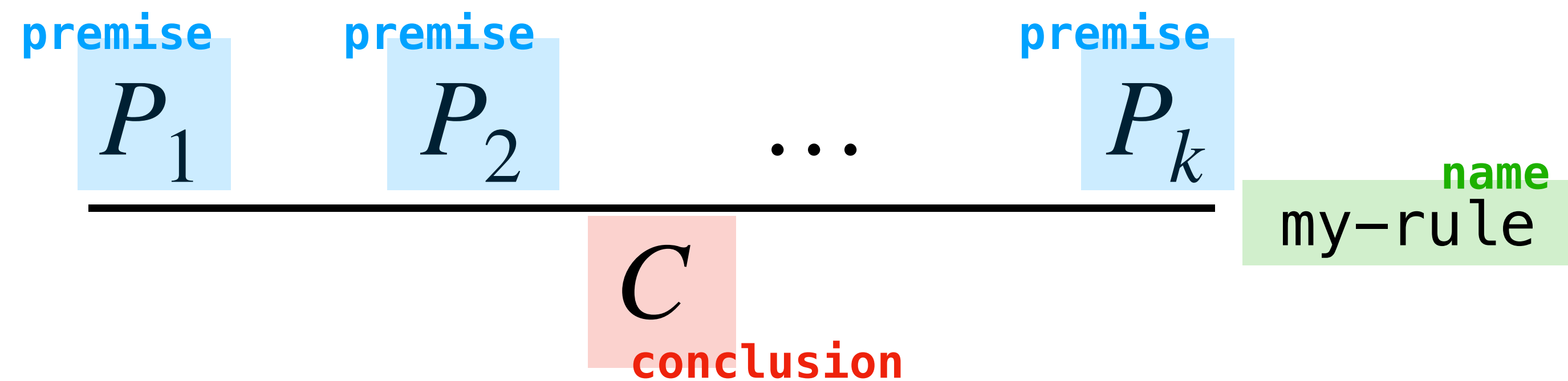
Inference Rules



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 through P_k hold, then C holds (by **my-rule**)*

Judgements

- » Syntax judgments
- » Typing judgments
- » Semantic judgments

Judgements

- » Syntax judgments
- » Typing judgments
- » Semantic judgments

Syntax Judgements

$$e \in WF$$

Syntax Judgements

$$e \in WF$$

A syntax judgement expresses that:

Syntax Judgements

$$e \in WF$$

A syntax judgement expresses that:

e is a well-formed expression

Syntax Judgements

$$e \in WF$$

A syntax judgement expresses that:

e is a well-formed expression

IMPORTANT: We will never use this kind of judgments explicitly!

Production Rules

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

$$\frac{e_1 \in \text{WF} \quad e_2 \in \text{WF}}{e_1 + e_2 \in \text{WF}} \quad (\text{addIntS})$$

Instead it's standard to use **production rules**

(We'll spend more time on this when we cover formal grammar)

Judgements

- » Syntax judgments
- » **Typing judgments**
- » Semantic judgments

Typing Judgments

$$\text{context } \Gamma \vdash \text{expression } e : \text{type } \tau$$

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

e is of type τ in the context Γ

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

$$\{ x: \text{int}, y: \text{int} \} \vdash x + x: \text{int}$$

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

Recall: 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 is an **int** (in any context Γ) and e_2 is an **int** then (in any context Γ) $e_1 + e_2$ is an **int** (in any context Γ)*

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 τ "*

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 τ "*

A context keeps track of all the types of variables in the *static* environment

Example: Reading Typing Judgements

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

Example: Reading Typing Judgements

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

In English: *Given that b is a bool , the expression $\text{if } b \text{ then } 2 \text{ else } 3$ is an int*

Example: Reading Typing Judgements

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

In English: *Given that b is a bool , the expression $\text{if } b \text{ then } 2 \text{ else } 3$ is an int*

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

Judgements are claims, not truths

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

Judgements are claims, not truths

```
{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

Judgements are claims, not truths

```
{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

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

Judgements are claims, not truths

```
{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

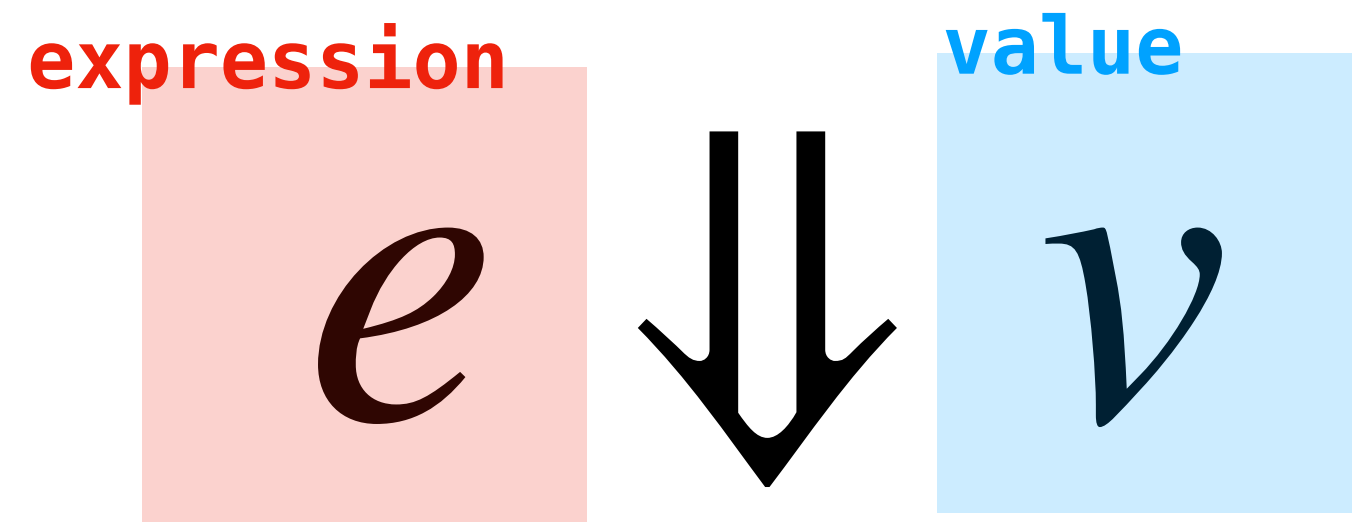
We haven't **proved** anything by writing down a typing judgment

Next time: we'll talk about **typing derivations**, which are used to demonstrate that expressions *actually* have their desired types in our PL

Judgements

- » Syntax judgments
- » Typing judgments
- » **Semantic judgments**

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

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) $v_1 + v_2$

Example: Reading Semantic Judgments

```
if 2 > 3 then 2 + 2 else 3 ↓↓ 3
```

In English: The expression

```
if 2 > 3 then 2 + 2 else 3
```

evaluates to the value 3

Values are not (Necessarily) Expressions

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

In this course, we will draw a distinction between values and expressions (note the font)

Example. We'll use regular numbers to represent integer values, and we'll use `T` and `⊥` for the true and false Boolean values

320Caml Inference Rules

Reminder: for every expression in our language, we given inference rules for syntax, typing, and semantics

Expressions

» Let-expressions

» If-Expressions

» Functions

» Application

Expressions

» **Let-expressions**

» If-Expressions

» Functions

» Application

Let-Expressions (Syntax Rule)

$$x \in \text{WF}(\text{Var}) \quad e_1 \in \text{WF}(\text{expr}) \quad e_2 \in \text{WF}(\text{expr})$$

$$\text{let } x = e_1 \text{ in } e_2 \in \text{WF}(\text{expr})$$

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-Expressions (Syntax Rule)

$\langle \text{expr} \rangle ::= \text{let } \overbrace{\langle \text{var} \rangle}^{\text{alphanumeric + '_' + '.'}} = \langle \text{expr} \rangle \text{ in } \langle \text{expr} \rangle$

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-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-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 τ_1 in the context Γ , and e_2 is of type τ in the context Γ with the variable declaration $(x : \tau_1)$ added to it, then

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

is of type τ in the context Γ

$$\vdash \underbrace{2}_{e_1} : \underbrace{\text{int}}_{\tau_1}$$

$$\vdash_{x:\text{int}} \underbrace{x + x}_{e_2} : \underbrace{\text{int}}_{\tau}$$

$$\vdash \text{let } x = \underbrace{2}_{e_1} \text{ in } \underbrace{x + x}_{e_2} : \underbrace{\text{int}}_{\tau}$$

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 τ_1 in the context Γ , and e_2 is of type τ in the context Γ *with the variable declaration* $(x : \tau_1)$ *added to it*, then

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

is of type τ in the context Γ

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 τ_1 in the context Γ , and e_2 is of type τ in the context Γ *with the variable declaration* $(x : \tau_1)$ *added to it*, then

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

is of type τ in the context Γ

Let-Expressions (Semantic Rule)

$e_1 \Downarrow v_1$

$[v_1 / x] e_2 \Downarrow v$

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

If e_1 evaluates to v_1 and e_2 with $\underset{\uparrow}{v_1}$ substituted for x evaluates to v , then

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

evaluates to v

$$\begin{array}{ccc}
 & 2+2 & \\
 & \text{''} & \\
 z \Downarrow 2 & [2/x](x+x) & \Downarrow 4
 \end{array}$$

$$\text{let } x = 2 \text{ in } x+x \Downarrow 4$$

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

Expressions

» Let-expressions

» **If-Expressions**

» Functions

» Application

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-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)

 $\Gamma \vdash e_1 : \text{bool}$ $\Gamma \vdash e_2 : \tau$ $\Gamma \vdash e_3 : \tau$

 $\Gamma \vdash \text{if } e, \text{ then } e_2 \text{ else } e_3 : \tau$

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 Γ and e_2 and e_3 are of type τ in the context Γ , then

`if e_1 then e_2 else e_3`

is of type τ in the context Γ

If-Expressions (Semantics)

$$e_1 \Downarrow T \qquad e_2 \Downarrow v_2$$

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_2$$
$$e_1 \Downarrow \perp$$
$$e_3 \Downarrow v_3$$

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_3$$

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

Expressions

» Let-expressions

» If-Expressions

» **Functions**

» Application

Functions (Syntax Rule)

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)

$$\frac{\{x:\text{int}\} \vdash x+x:\text{int}}{\{\} \vdash \text{fun } x \rightarrow x+x:\text{int} \rightarrow \text{int}}$$

$$\Gamma, x:\text{int} \vdash e:\tau_2$$

$$\Gamma \vdash \text{fun } x \rightarrow e:\tau_1 \rightarrow \tau_2$$

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 τ_2 in the context Γ 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 Γ

Functions (Semantic Rule)

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)

Expressions

» Let-expressions

» If-Expressions

» Functions

» **Application**

Application (Syntax Rule)

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)

$$\frac{\Gamma \vdash e_1 : \boxed{\tau_2} \Rightarrow \boxed{\tau} \quad \Gamma \vdash e_2 : \boxed{\tau_2}}{\Gamma \vdash e_1 e_2 : \boxed{\tau}}$$

If e_1 has type $\tau_2 \rightarrow \tau$ under the context Γ and e_2 is of type τ_2 under the context Γ , then $e_1 e_2$ is of type τ under the context Γ

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 Γ and e_2 is of type τ_2 under the context Γ , then $e_1 e_2$ is of type τ under the context Γ

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

Example (Informal)

```
(let x = 2 in fun y -> x + y) (2 + 3)
```

We 'll see more typing
rules and semantic rules

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

Summary

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