Inference Rules

Concepts of Programming Languages
Lecture 5

CAS CS 320

Practice Problem

Determine an expression with the following type

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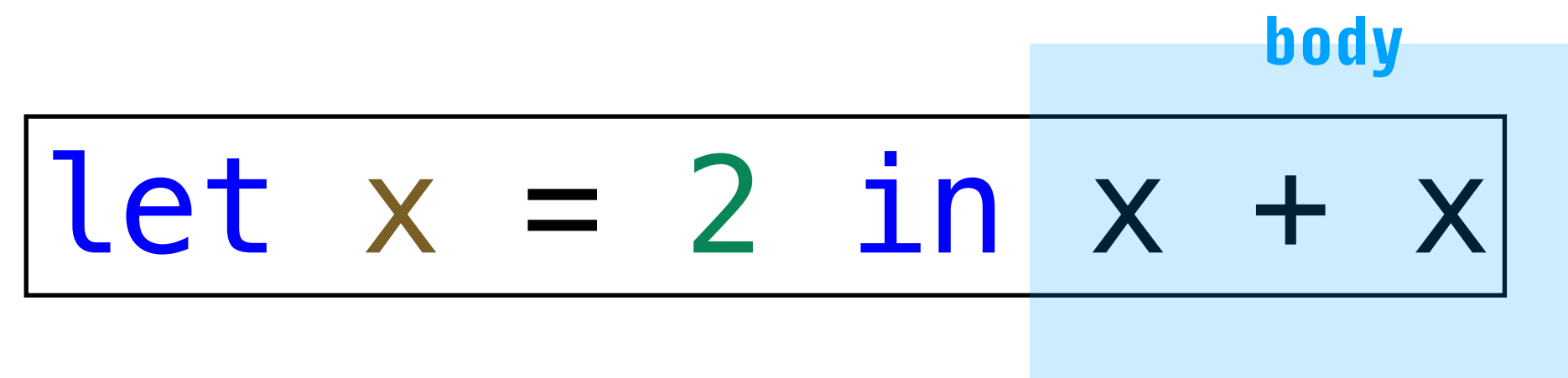
Outline

» 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
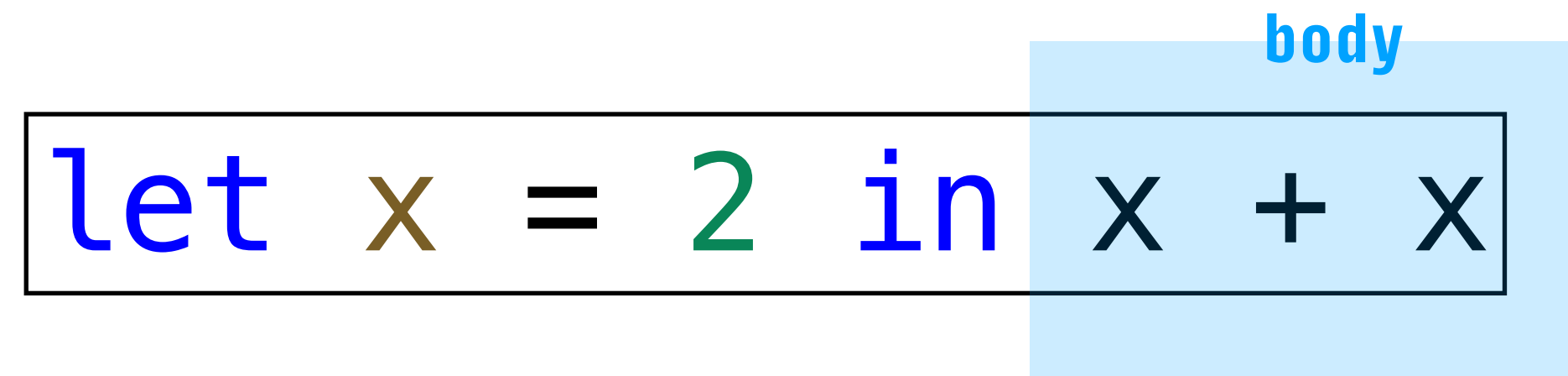
Recap

Recall: Local Variables (Informal)

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

body
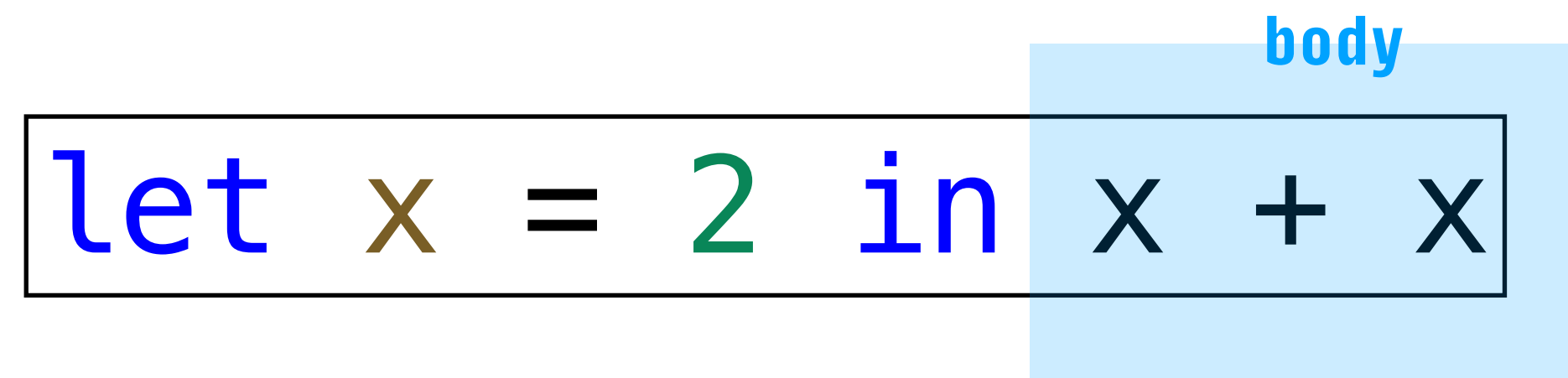
Recall: Local Variables (Informal)

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

Syntax: let VARIABLE = EXPRESSION in BODY

Recall: Local Variables (Informal)

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

body

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)

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

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*

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

Recall: Functions (Informal)

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

Syntax: fun VAR-NAME -> EXPR

Recall: Functions (Informal)
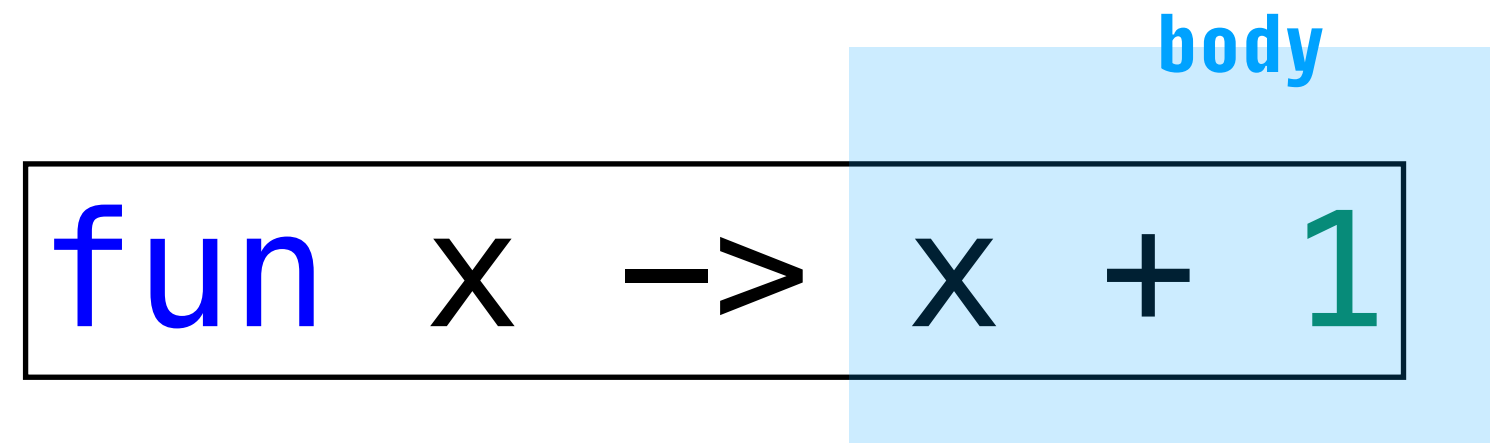
body

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

Syntax: fun VAR-NAME -> EXPR

Typing: the type of a function is T1 -> T2 where T1 is the type of the input and T2 is the type of the output

Recall: Functions (Informal)

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

Syntax: fun VAR-NAME -> EXPR

Typing: the type of a function is T1 -> 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 -> 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 -> 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

Inference Rules

Inference Rules
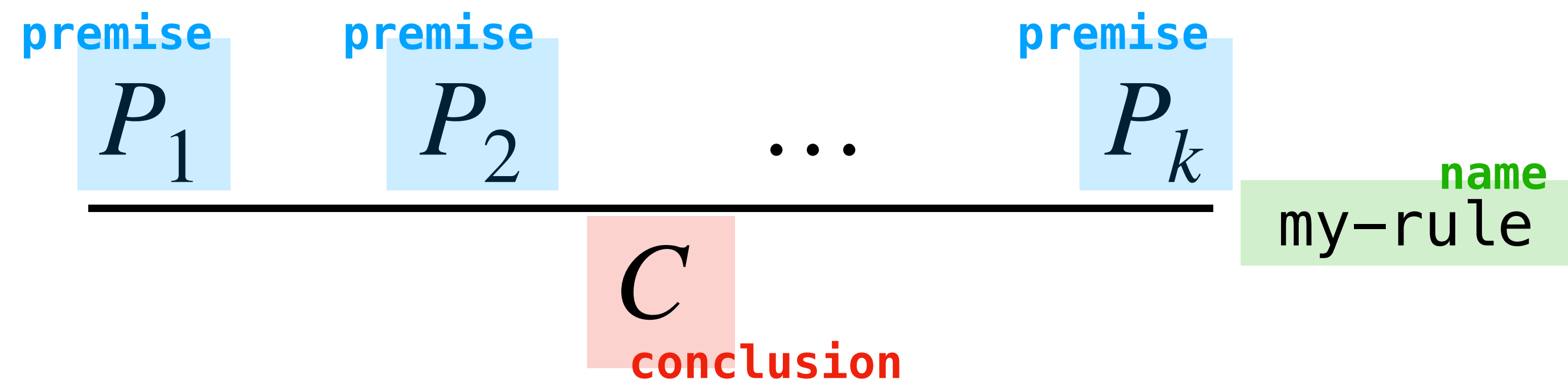
$$\frac{P_1 \quad P_2 \quad \dots \quad P_k}{C} \text{ my-rule}$$

premise P_1

premise P_2

premise P_k

C conclusion

name my-rule

Inference Rules

$$\frac{P_1 \quad P_2 \quad \dots \quad P_k}{C} \text{ my-rule}$$

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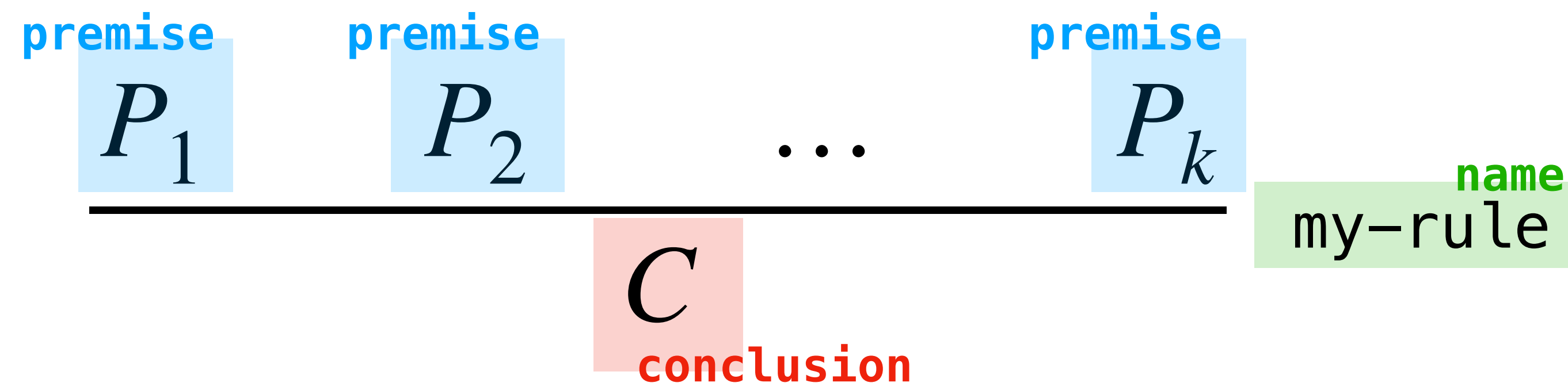
Then general form of an inference rule has a collection of **premises** and a **conclusion**

Inference Rules

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_k}{C} \text{ my-rule}$$

premise P_1, premise P_2, premise P_k, name my-rule, conclusion C
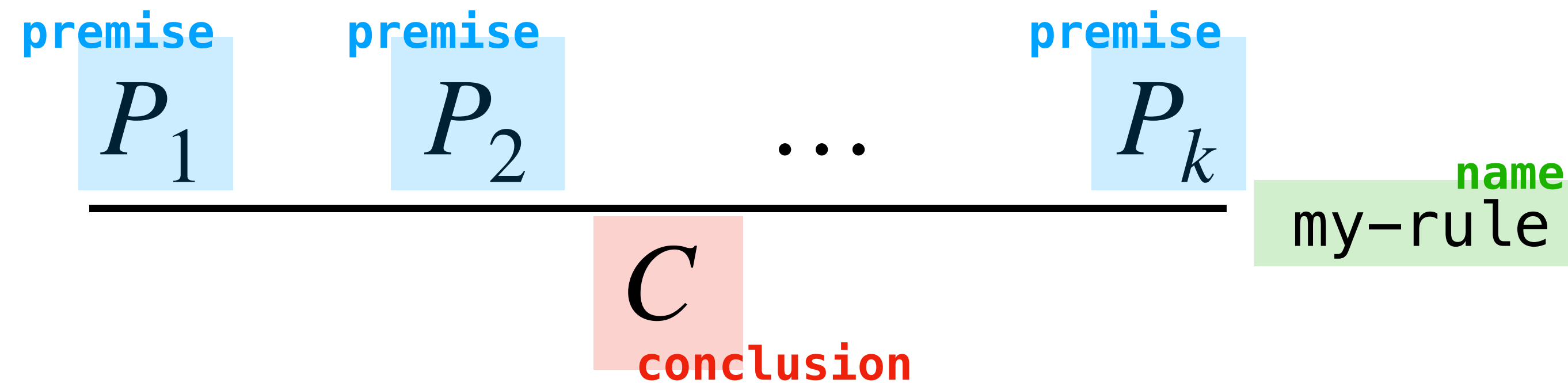
Then general form of an inference rule has a collection of **premises** and a **conclusion**

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

Inference Rules

$$\frac{P_1 \quad P_2 \quad \dots \quad P_k}{C} \text{ my-rule}$$

premise P_1 premise P_2 premise P_k name my-rule

conclusion C
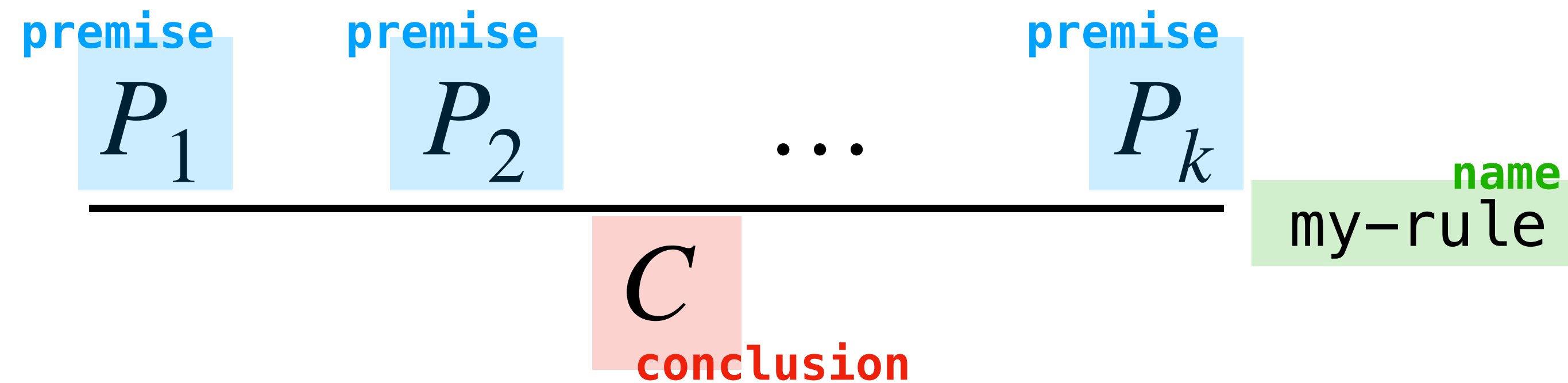
We can read this as:

*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

$$e \in \mathsf{WF}$$

Syntax Judgements

$$e \in \mathsf{WF}$$

A syntax judgement expresses that:

Syntax Judgements

$$e \in \mathsf{WF}$$

A syntax judgement expresses that:

e is a well-formed expression

Syntax Judgements

$$e \in \mathsf{WF}$$

A syntax judgement expresses that:

e is a well-formed expression

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

Production Rules

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

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

$$\Gamma \vdash e : \tau$$

A <u>typing judgment</u> a compact way of representing the statement:

$$e \text{ is of type } \tau \text{ in the context } \Gamma$$

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

Recall: Integer Addition Typing Rule

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

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

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

Contexts

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

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

Contexts

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

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

$$\Gamma = \{ \ \texttt{x : int}, \ \texttt{y : string}, \ \texttt{z : int -> string} \ \}$$

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

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

Example: Reading Typing Judgements

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

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

Example: Reading Typing Judgements

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

In English: *Given that b is a bool, the expression if b then 2 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

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

Judgements are claims, not truths

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

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

Judgements are claims, not truths

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

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

Judgements are claims, not truths

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

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
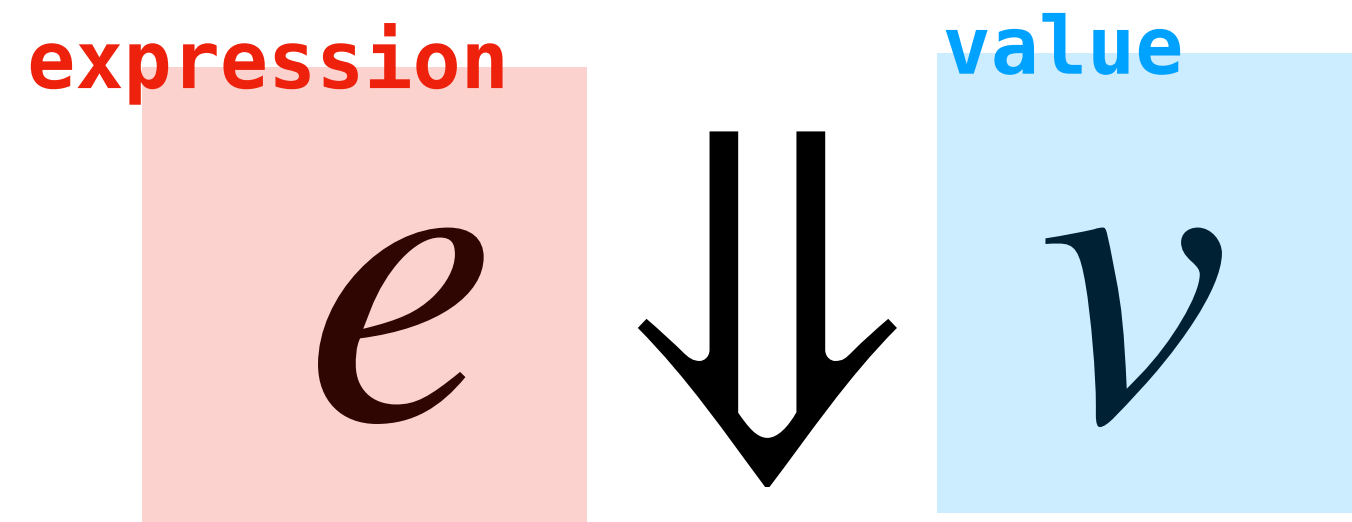
$$\underset{\text{expression}}{e} \Downarrow \underset{\text{value}}{v}$$

A <u>semantic judgment</u> 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 judgments

Recall: Integer Addition Semantic Rule

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \; \textcolor{red}{+} \; e_2 \Downarrow v_1 + v_2} \; \text{(evalInt)}$$

If e_1 evaluates to the (integer) v_1 and e_2 evaluates to the (integer) v_2, then e_1 $\textcolor{red}{+}$ e_2 evaluates to the (integer) $v_1 + v_2$

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

Values are not (Necessarily) Expressions

if 2 > 3 then 2 + 2 else 3 ⇓ 3

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

Example. We'll use regular numbers to represented integer values, and we'll use ⊤ 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)

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

<p style="color:red; text-align:center;">let x = e_1 in e_2</p>

is a well–formed expression

Let-Expressions (Syntax Rule)

<expr> ::= let <var> = <expr> in <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 (Typing Rule)

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

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

is of type τ in the context Γ

Let-Expressions (Typing Rule)

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

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

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

is of type τ in the context Γ

Let-Expressions (Typing Rule)

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

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)

If e_1 evaluates to v_1 and e_2 *with* v_2 *substituted for* x evaluates to v, then

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

evaluates to v

Let-Expressions (Semantic Rule)

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

If e_1 evaluates to v_1 and e_2 *with* v_2 *substituted for* x evaluates to v, then

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

evaluates to v

Expressions

» Let-expressions

» **If-Expressions**

» Functions

» Application

If-Expressions (Syntax Rule)

If-Expressions (Syntax Rule)

`<expr> ::= if <expr> then <expr> else <expr>`

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)

If-Expressions (Typing Rule)

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

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

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

is of type τ in the context Γ

If-Expressions (Semantics)

If-Expressions (Semantic Rule 1)

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

If e_1 evaluates to $\top$ 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)

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

If e_1 evaluates to $\bot$ 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)

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

Note: we never evaluate both branches

If e_1 evaluates to $\bot$ 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)

```
<expr> ::= fun <var> -> <expr>
```

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

$$\text{fun } x \text{ -> } e$$

is a well-formed expression

Functions (Typing Rule)

Functions (Typing Rule)

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

If e has type τ_2 in the context Γ with the variable declaration $(x : \tau_1)$ added, then

$$\texttt{fun } x \texttt{ -> } e$$

is of type $\tau_1 \texttt{ -> } \tau_2$ in the context Γ

Functions (Semantic Rule)

Functions (Semantic Rule)

$$\frac{}{\texttt{fun } x \texttt{ -> } e \Downarrow \lambda x.e} \text{ (funEval)}$$

Under no premises, the expression

$$\texttt{fun } x \texttt{ -> } e$$

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)

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

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)

If e_1 has type τ_2 **->** τ 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)

$$\frac{\Gamma \vdash e_1 : \tau_2 \ \textcolor{red}{\text{->}}\ \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau} \ \text{(app)}$$

If e_1 has type τ_2 $\textcolor{red}{\text{->}}$ τ 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)

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

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