

# Quiz

- Question #1: What is the value of the following expression?

$+(1,1)$

- Wrong answer: **0**
- Wrong answer: **42**
- Answer: **2**

# Quiz

- Question #2: What is the value of the following expression?  
**+ proc 8**
- Wrong answer: **error**
- Answer: Trick question! **+ proc 8** is not an expression

# Language Grammar for Quiz

```
<expr>   ::=   <num>
           ::=   <bool>
           ::=   <id>
           ::=   <prim> ( { <expr> }*(,) )
           ::=   proc (<id>*(,)) <expr>
           ::=   (<expr> <expr>*)
           ::=   if <expr> then <expr> else <expr>
<prim>   ::=   +   |   -   |   *   |   add1   |   sub1
<bool>   ::=   true   |   false
```

# Quiz

- Question #3: Is the following an expression?

**add1(1, 7)**

- Wrong answer: **No**
- Answer: **Yes** (according to our grammar)

# Quiz

- **Question #4:** What is the value of the following expression?  
**add1(1, 7)**
- **Answer:** **2** (according to our interpreter)
- But no *real* language would accept **add1(1, 7)**
- Let's agree to call **add1(1, 7)** an ***ill-formed expression*** because **add1** should be used with only one argument
- Let's agree to never evaluate ill-formed expressions

# Quiz

- **Question #5:** What is the value of the following expression?

**add1(1, 7)**

- **Answer:** **None** - the expression is ill-formed

# Quiz

- Question #6: Is the following a well-formed expression?

$+(\mathbf{proc}(x)x, 5)$

- Answer: Yes

# Quiz

- **Question #7:** What is the value of the following expression?

`+(proc(x)x, 5)`

- **Answer:** **None** - it produces an error:

`+`: expects type <number> as 1st argument,  
given: (closure ((cbv-var x)) (var-exp x)  
(empty-env-record)); other arguments were: 5

- Let's agree that a **proc** expression cannot be inside a `+` form

# Quiz

- Question #8: Is the following a well-formed expression?

`+(proc(x)x, 5)`

- Answer: No

# Quiz

- **Question #9:** Is the following a well-formed expression?  
 $+((\mathbf{proc}(x)x\ 7),\ 5)$
- **Answer:** Depends on what we meant by *inside* in our most recent agreement
  - *Anywhere inside* - **No**
  - *Immediately inside* - **Yes**
- Since our interpreter produces **12**, and since that result makes sense, let's agree on *immediately inside*

# Quiz

- Question #10: Is the following a well-formed expression?  
 $+((\mathbf{proc}(x)x \, \mathbf{true}), \, 5)$
- Answer: Yes, but we don't want it to be!

# Quiz

- Question #11: Is it possible to define ***well-formed*** (as a decidable property) so that we reject all expressions that produce errors?
- Answer: Yes: reject *all* expressions!

## Quiz

- Question #12: Is it possible to define ***well-formed*** (as a decidable property) so that we reject *only* expressions that produce errors?
- Answer: No

**+ (1, if ... then 1 else proc(x)x)**

- If we always knew whether ... produces true or false, we could solve the halting problem

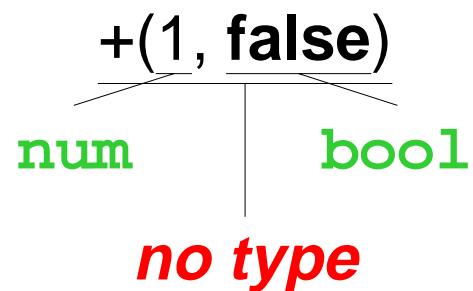
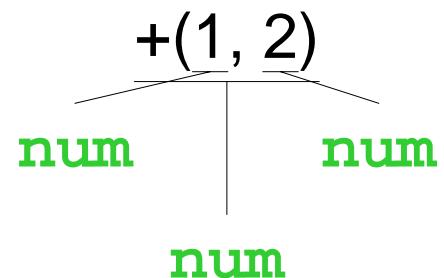
# Types

- Solution to our dilemma
  - In the process of rejecting expressions that are certainly bad, also reject some expressions that are good
    - +**(1, if (prime? 131101) then 1 else proc(x)x)**
- Overall strategy:
  - Assign a **type** to each expression *without evaluating*
  - Compute the type of a complex expression based on the types of its subexpressions

# Types

1 : num

true : bool



# Type Rules

`<num> : num`

`<bool> : bool`

`<expr>1 : num      <expr>2 : num`

`+(<expr>1, <expr>2) : num`

`1 : num`

`true : bool`

`1 : num      2 : num`

`+(<expr>1, <expr>2) : num`

`1 : num`

`false : bool`

`+(<expr>1, false) : no type`

# Type Rules

`<num> : num`

`<bool> : bool`

$\frac{}{<\text{expr}>_1 : \text{num}}$        $\frac{}{<\text{expr}>_2 : \text{num}}$

$+(<\text{expr}>_1, <\text{expr}>_2) : \text{num}$

`1 : num`

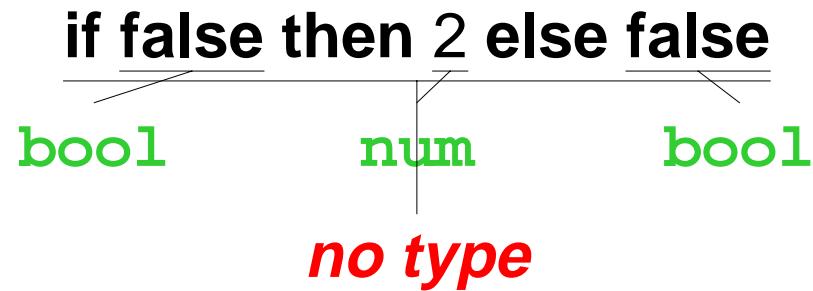
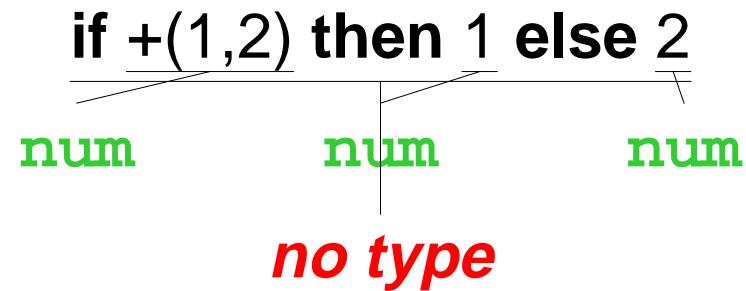
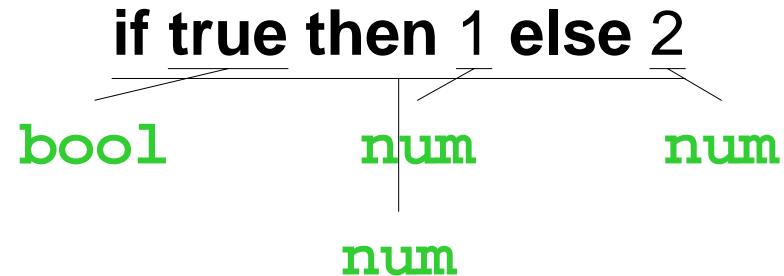
`2 : num`

$\frac{}{+(1, 2) : \text{num}}$

`3 : num`

$\frac{}{+((1, 2), 3) : \text{num}}$

# Types: Conditionals



# Conditional Type Rules

$\langle \text{expr} \rangle_1 : \text{bool}$

$\langle \text{expr} \rangle_2 : \langle \text{type} \rangle_0$

$\langle \text{expr} \rangle_3 : \langle \text{type} \rangle_0$

---

**if**  $\langle \text{expr} \rangle_1$  **then**  $\langle \text{expr} \rangle_2$  **else**  $\langle \text{expr} \rangle_3 : \langle \text{type} \rangle_0$

**true** :  $\text{bool}$

$1 : \text{num}$

$2 : \text{num}$

---

**if true then 1 else 2 : num**

$+(1,2) : \text{num}$

$1 : \text{num}$

$2 : \text{num}$

---

**if  $+(1,2)$  then 1 else 2 : no type**

**false** :  $\text{bool}$

$2 : \text{num}$

**false** :  $\text{bool}$

---

**if false then 2 else false : no type**

# Types: Variables and Functions

$x : \text{no type}$

$\text{proc}(\text{bool } x)x$

bool

$(\text{bool} \rightarrow \text{bool})$

$\text{proc}(\text{bool } x)\text{if } x \text{ then } 1 \text{ else } 2$

bool

num

num

num

$(\text{bool} \rightarrow \text{num})$

# Variable and Function Type Rules

$$\{ \dots \langle \text{id} \rangle : T \dots \} \vdash \langle \text{id} \rangle : T$$

$$\frac{\{ \langle \text{id} \rangle : T_1 \} + E \vdash e : T_2}{E \vdash \text{proc}(T_1 \langle \text{id} \rangle) e : (T_1 \rightarrow T_2)}$$

Abbreviations:  $T = \langle \text{type} \rangle$      $e = \langle \text{expr} \rangle$      $E = \langle \text{env} \rangle$

# Variable and Function Type Rules

$$\{ \dots \langle \text{id} \rangle : T \dots \} \vdash \langle \text{id} \rangle : T$$

$$\frac{\{ \langle \text{id} \rangle : T_1 \} + E \vdash e : T_2}{E \vdash \text{proc}(T_1 \langle \text{id} \rangle) e : (T_1 \rightarrow T_2)}$$

$$\{ \} \vdash x : \text{no type}$$

$$\frac{\{x : \text{bool}\} \vdash x : \text{bool}}{\{ \} \vdash \text{proc(bool } x\text{)} x : (\text{bool} \rightarrow \text{bool})}$$

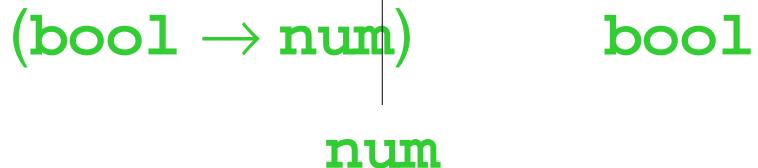
$$\frac{\begin{array}{c} \{x : \text{bool}\} \vdash x : \text{bool} \quad \{x : \text{bool}\} \vdash 1 : \text{num} \quad \{x : \text{bool}\} \vdash 2 : \text{num} \\ \hline \{x : \text{bool}\} \vdash \text{if } x \text{ then } 1 \text{ else } 2 : \text{num} \end{array}}{\{ \} \vdash \text{proc(bool } x\text{)} \text{if } x \text{ then } 1 \text{ else } 2 : (\text{bool} \rightarrow \text{num})}$$

# Revised Rules

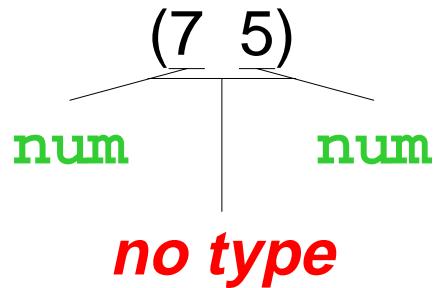
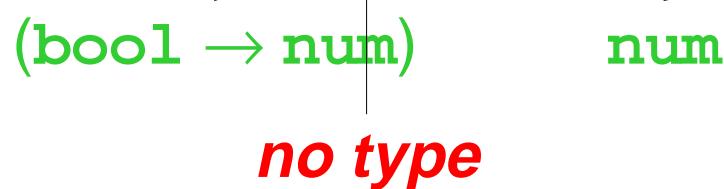
$$E \vdash \langle \text{num} \rangle : \text{num}$$
$$E \vdash \langle \text{bool} \rangle : \text{bool}$$
$$E \vdash e_1 : \text{num}$$
$$E \vdash e_1 : \text{num}$$
$$E \vdash +(e_1, e_2) : \text{num}$$
$$E \vdash e_1 : \text{bool}$$
$$E \vdash e_2 : T_0$$
$$E \vdash e_3 : T_0$$
$$E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T_0$$

# Types: Function Calls

(proc(bool x)if x then 1 else 2 true)



(proc(bool x)if x then 1 else 2 5)



# Function Call Type Rule

$$\frac{E \vdash e_1 : (T_2 \rightarrow T_3) \quad E \vdash e_2 : T_2}{E \vdash (e_1 e_2) : T_3}$$

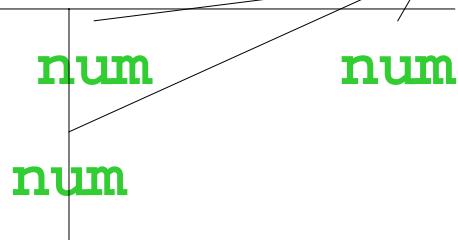
$$\frac{\{ \} \vdash \text{proc(bool } x \text{)if } x \text{ then } 1 \text{ else } 2 : (\text{bool} \rightarrow \text{num}) \quad \{ \} \vdash \text{true} : \text{bool}}{\{ \} \vdash (\text{proc(bool } x \text{)if } x \text{ then } 1 \text{ else } 2 \ \text{ true}) : \text{num}}$$

$$\frac{\{ \} \vdash \text{proc(bool } x \text{)if } x \text{ then } 1 \text{ else } 2 : (\text{bool} \rightarrow \text{num}) \quad \{ \} \vdash 5 : \text{num}}{\{ \} \vdash (\text{proc(bool } x \text{)if } x \text{ then } 1 \text{ else } 2 \ 5) : \text{no type}}$$

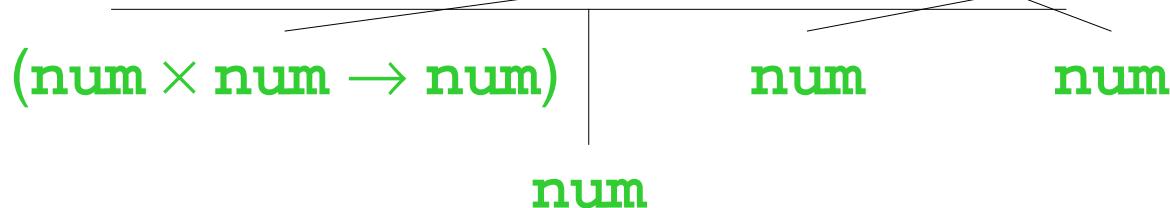
$$\frac{\{ \} \vdash 7 : \text{num} \quad \{ \} \vdash 5 : \text{num}}{\{ \} \vdash (7 \ 5) : \text{no type}}$$

# Types: Multiple Arguments

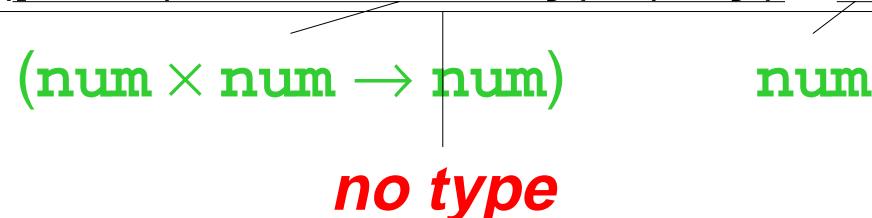
proc(num x, num y)+(x, y)



(proc(num x, num y)+(x, y) 5 6)



(proc(num x, num y)+(x, y) 5)



## Revised Function and Call Rules

$$\frac{\{ \langle \text{id} \rangle_1 : T_1, \dots \langle \text{id} \rangle_n : T_n \} + E \vdash e : T_0}{E \vdash \text{proc}(T_1 \langle \text{id} \rangle_1, \dots T_n \langle \text{id} \rangle_n) e : (T_1 \times \dots T_n \rightarrow T_0)}$$

$$\frac{E \vdash e_0 : (T_1 \times \dots T_n \rightarrow T_0) \quad E \vdash e_1 : T_1 \quad \dots \quad E \vdash e_n : T_n}{E \vdash (e_0 \ e_1 \ \dots \ e_n) : T_0}$$

# New Interpreter and Checker

- Change our interpreter:
  - Add types for arguments and letrec results to the grammar
- Implement a type-checker:
  - Produces the same type that the rules procedure
  - Calls itself recursively to get types for sub-expressions
  - Treat primitives as built-in functions

$+ : (\mathbf{num} \times \mathbf{num} \rightarrow \mathbf{num})$