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.

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 (e,g., C++) 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?

```
+(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?

```
+((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 intrepreter produces **12**, and since that result makes sense, let’s agree on *immediately inside*.

Quiz

- Question #10: Is the following a well-formed expression?

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

- Answer: **Yes**, but we don’t want it to be!

Quiz

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

Quiz

- Question #12: Is it possible to define ***well-formed*** (as a decideable 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.

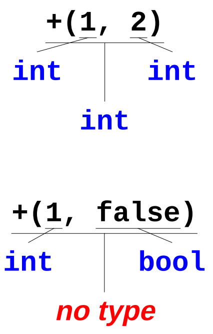
```
+(1, if (prime? 131101) then 1 else proc(x)x)
```

- Overall strategy:
  - Assign a ***type*** to each expression.
  - Compute the type of a complex expression based on the types of its subexpressions.

Types

`1 : int`

`true : bool`

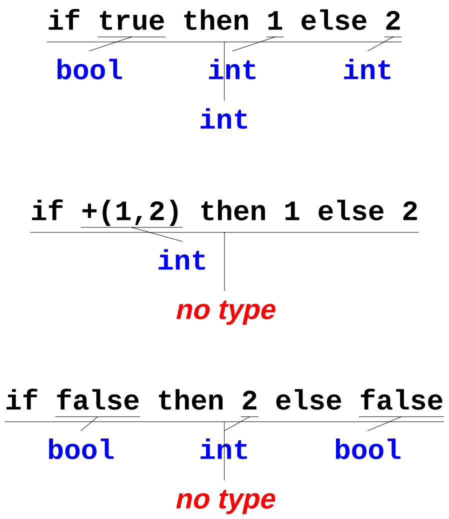


```
+(1, 2)
int    int
  int
```

```
+(1, false)
int    bool
```

no type

Types



```
if true then 1 else 2
bool    int    int
        int
```

```
if +(1,2) then 1 else 2
       int
```

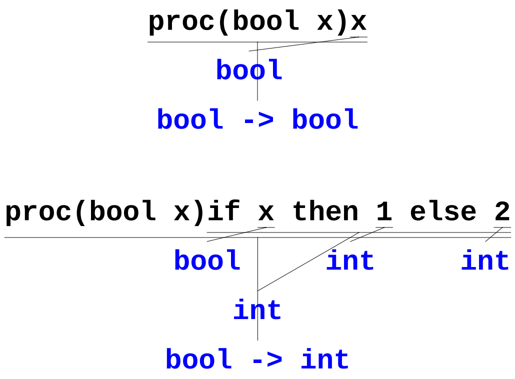
no type

```
if false then 2 else false
bool     int     bool
```

no type

Types

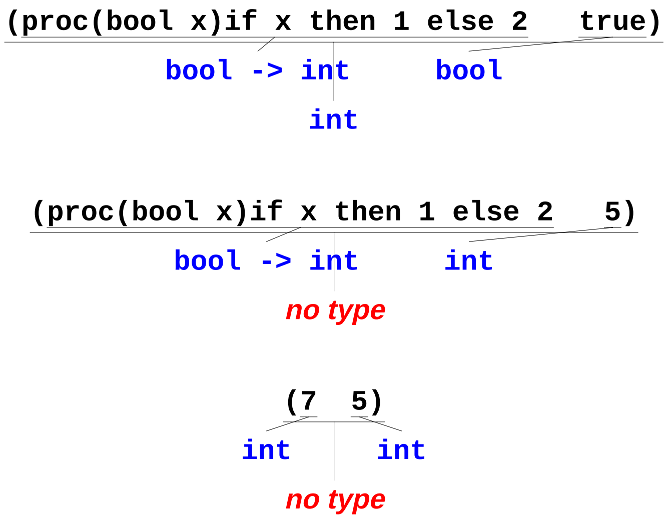
`x :` *no type*



```
proc(bool x)x
      bool
bool -> bool
```

```
proc(bool x)if x then 1 else 2
              bool   int    int
                  int
              bool -> int
```

Types

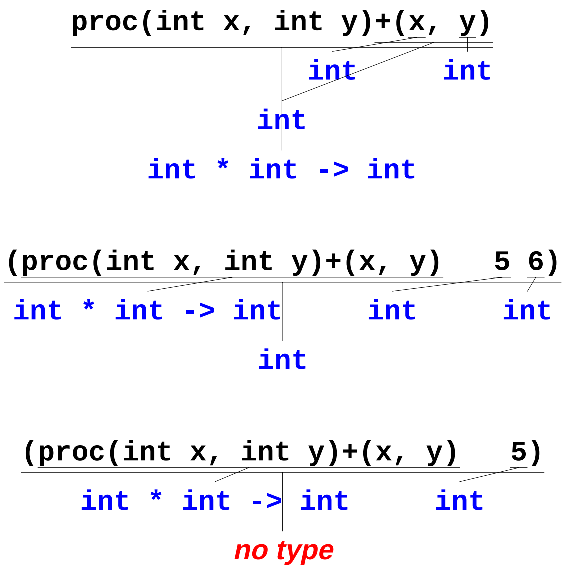


```
(proc(bool x)if x then 1 else 2    true)
        bool -> int       bool
                int
```

```
(proc(bool x)if x then 1 else 2    5)
        bool -> int       int
                no type
```

```
(7   5)
int      int
   no type
```

Types



```
proc(int x, int y)+(x, y)
              int     int
            int
      int * int -> int
```

```
(proc(int x, int y)+(x, y)    5 6)
int * int -> int       int      int
             int
```

```
(proc(int x, int y)+(x, y)    5)
int * int -> int        int
           no type
```

New Interpreter and Checker

- Change our interpreter:
  - ○ Add types for arguments and letrec results to the grammar
- Implement a type-checker:
  - ○ Recursively assign types to subexpressions
  - ○ Check consistency at **`if`** and application
  - ○ Treat primitives as built-in functions

```
+ : int * int -> int
```