# Type Checking

# Previously, in PL…

- **Scoping rules** match identifier uses with identifier definitions.
- A **type** is a set of values coupled with a set of operations on those values.
- A **type system** specifies which operations are valid for which types.
- **Type checking** can be done statically (at compile time) or dynamically (at run time).

# Today, in PL (one slide summary)

- A **type environment** gives types for free variables. You typecheck a let-body with an environment that has been **updated** to contain the new let-variable.

- If an object of type X could be used when one of type Y is acceptable then we say X is a **subtype** of Y, also written **X ≤ Y**.

- A type system is **sound** if

$$\forall E.\ \text{dynamic\_type}(E) \leq \text{static\_type}(E)$$

# Lecture Outline

- Typing Rules
- Typing Environments
- “Let” Rules
- Subtyping
- Wrong Rules

# Type Checking Proofs

- A type checker's goal is to make sure that the program follows the **rules** for the given language's **type system**.
- Type checking proves facts e : T (read as: the expression e has type T).
  - One type rule is used for each kind of expression
- In a type rule used for a node e:
  - The **hypotheses** are the proofs of types of e's subexpressions
  - The **conclusion** is the proof of the type of e itself (the fact that we're trying to prove, namely that e has type T).

# Review Example: 1 + 2

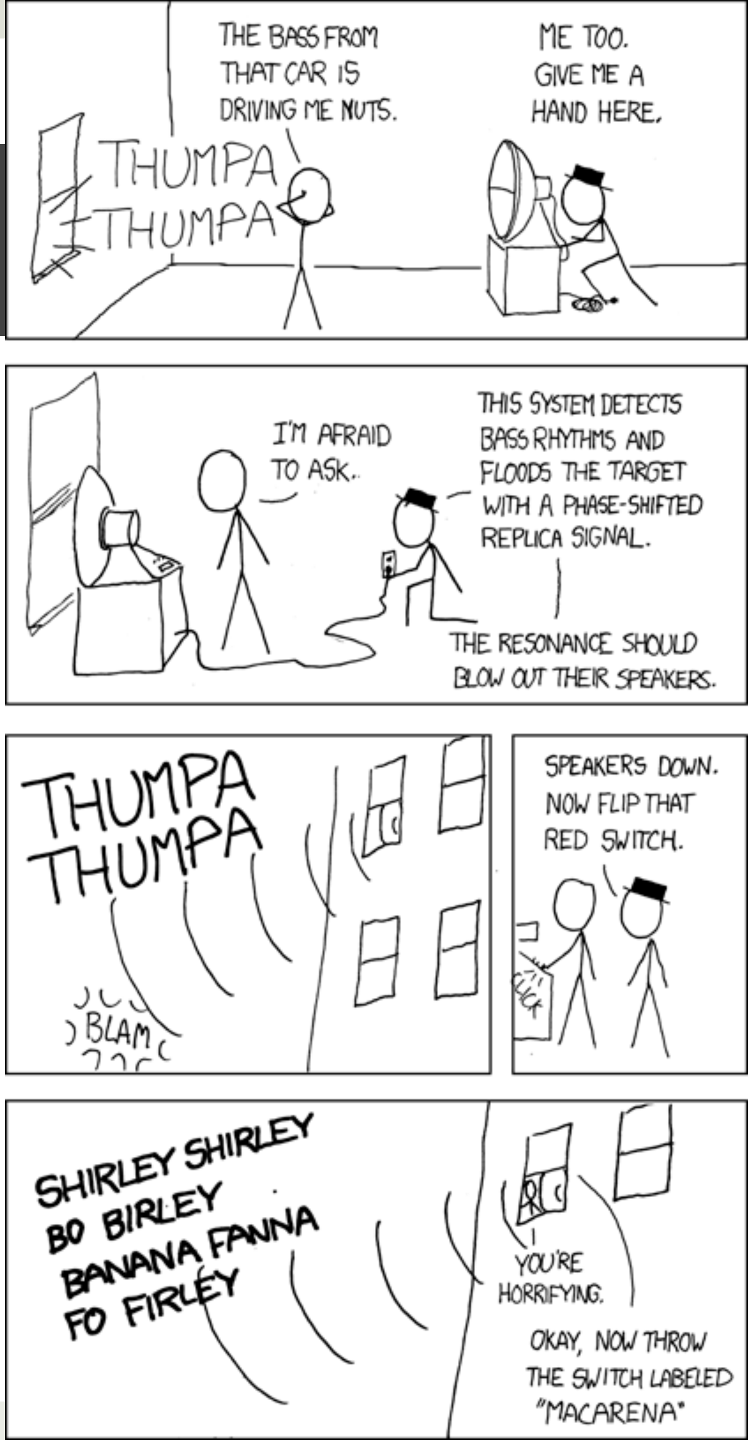
$$\dfrac{\dfrac{}{\vdash 1 : \mathrm{Int}} \qquad \dfrac{}{\vdash 2 : \mathrm{Int}}}{\vdash 1 + 2 : \mathrm{Int}}$$

If we can prove it, then it's true!

# Soundness

- A type system is **sound** if
  - Whenever $\vdash e : T$
  - Then $e$ evaluates to a value of type $T$
- We only want sound rules
  - But some sound rules are better than others:

$$\frac{(i \text{ is an integer})}{\vdash i : \text{Object}}$$



# Soundness: Why we care.

- If the type system is sound, then we can say that “a well-typed program won’t have runtime type mistakes.”
- Completeness is the converse of soundness: A logical system is **complete** if
  - Whenever $e$ evaluates to a value of type $T$
  - Then $\vdash e : T$
- There’s a tug-of-war, here; we’ll get back to this idea later.

# Rules for Constants

$$\frac{}{\vdash \text{false} : \text{Bool}} \; [\text{Bool}]$$

$$\frac{}{\vdash s : \text{String}} \; [\text{String}]$$

(s is a string constant)

# Rule for New

- new T produces an object of type T
  - Ignore SELF_TYPE for now . . .

$$\frac{}{\vdash \text{new } T : T} \quad [\text{New}]$$

# Two More Rules

$$\frac{\vdash e : \text{Bool}}{\vdash \text{not } e : \text{Bool}} \quad [\text{Not}]$$

$$\frac{\begin{array}{c}\vdash e_1 : \text{Bool} \\ \vdash e_2 : T\end{array}}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}} \quad [\text{Loop}]$$

# Two More Rules

$$\frac{\vdash e : \mathrm{Bool}}{\vdash \mathrm{not}\ e : \mathrm{Bool}}$$

What does a **type derivation** using this rule look like?

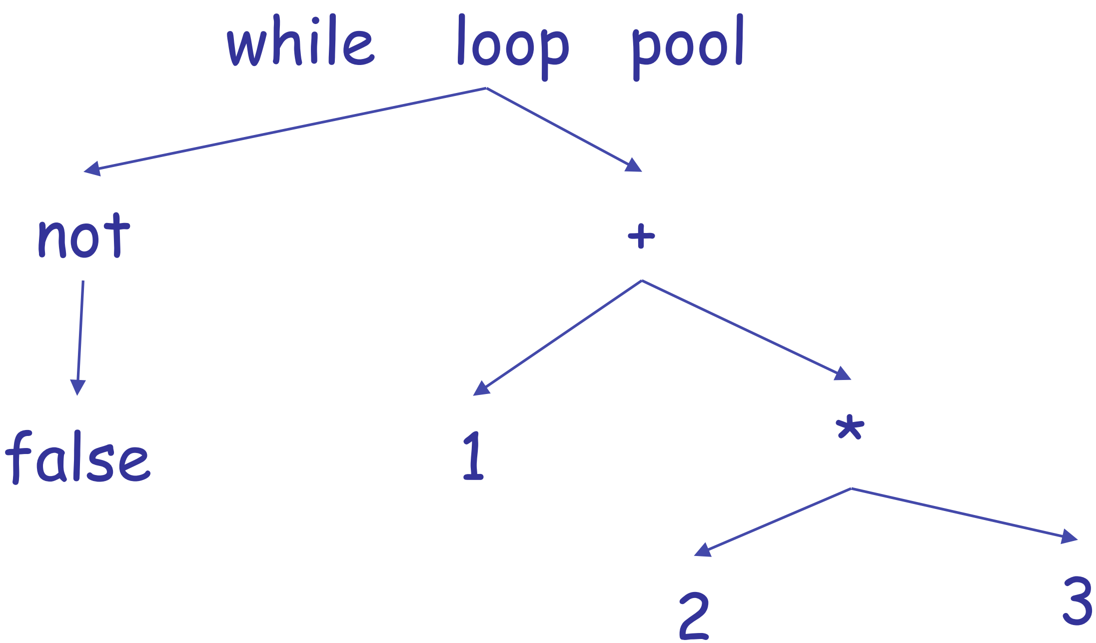
$$\frac{\vdash e_1 : \mathrm{Bool} \qquad \vdash e_2 : T}{\vdash \mathrm{while}\ e_1\ \mathrm{loop}\ e_2\ \mathrm{pool} : \mathrm{Object}}\ [\mathrm{Loop}]$$

# Typing: Example

**while not false loop 1 + 2 * 3 pool**

# Typing: Example

**while not false loop 1 + 2 * 3 pool**

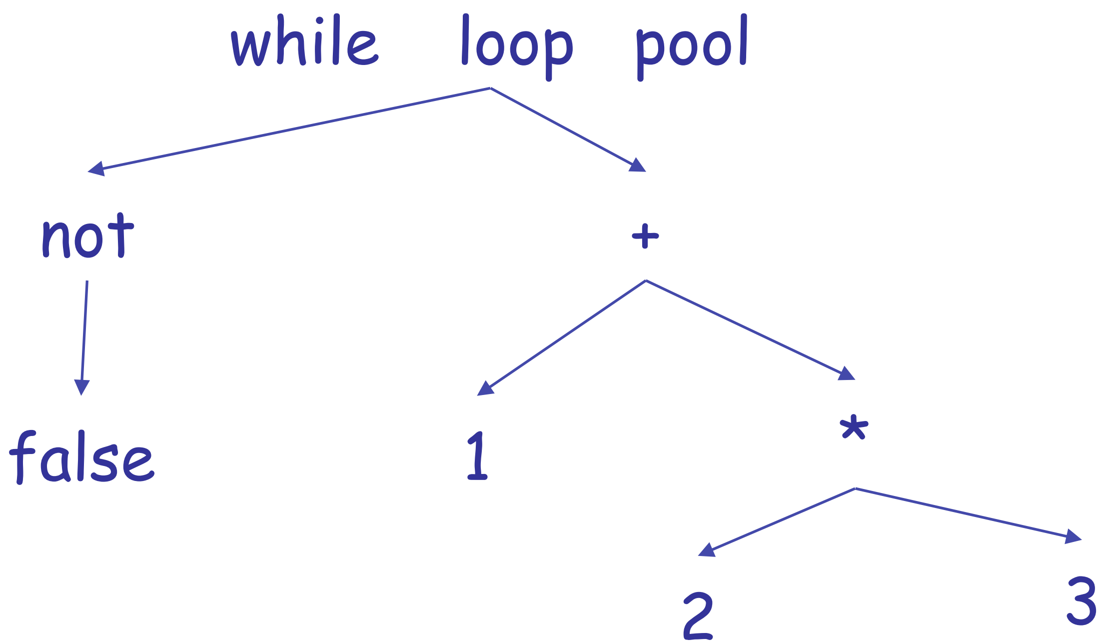


# Typing: Example

**while not false loop 1 + 2 * 3 pool**

**Loop rule:**

$$\frac{\vdash e_1 : \text{Bool} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}}$$

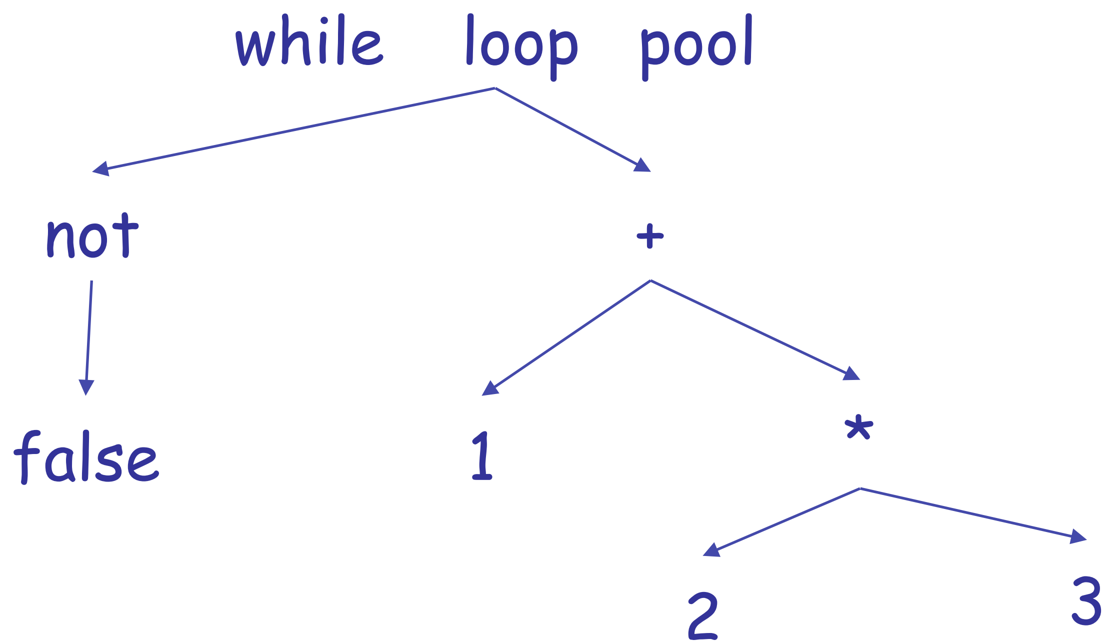


# Typing: Example

**while not false loop 1 + 2 * 3 pool**

**Loop rule:**

$$\frac{\vdash e_1 : \text{Bool} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}}$$

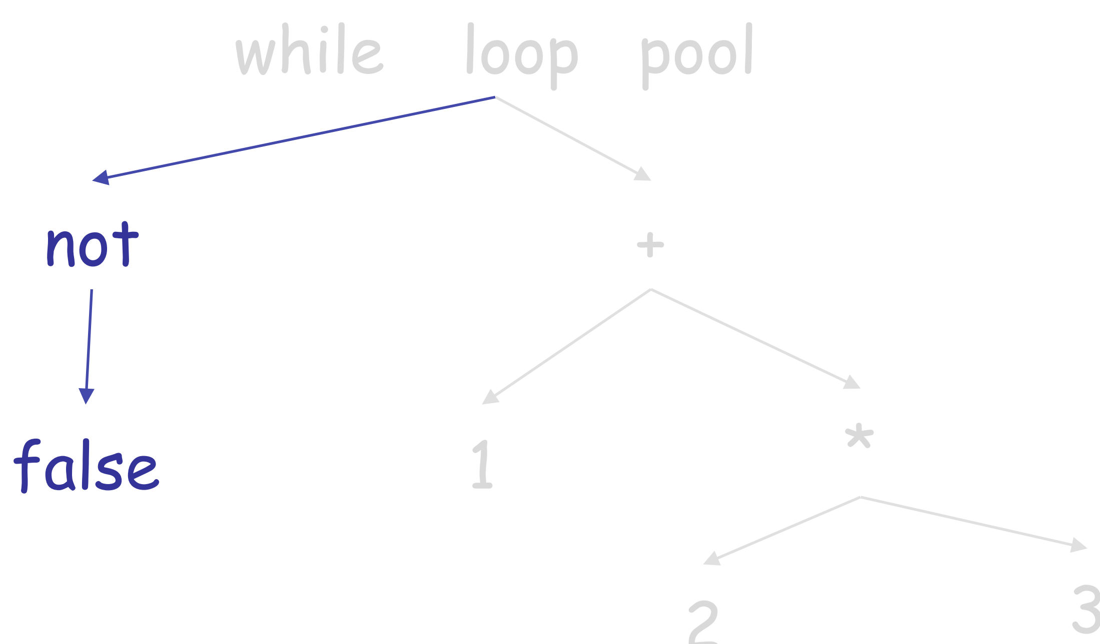


# Typing: Example

**while not false loop 1 + 2 * 3 pool**

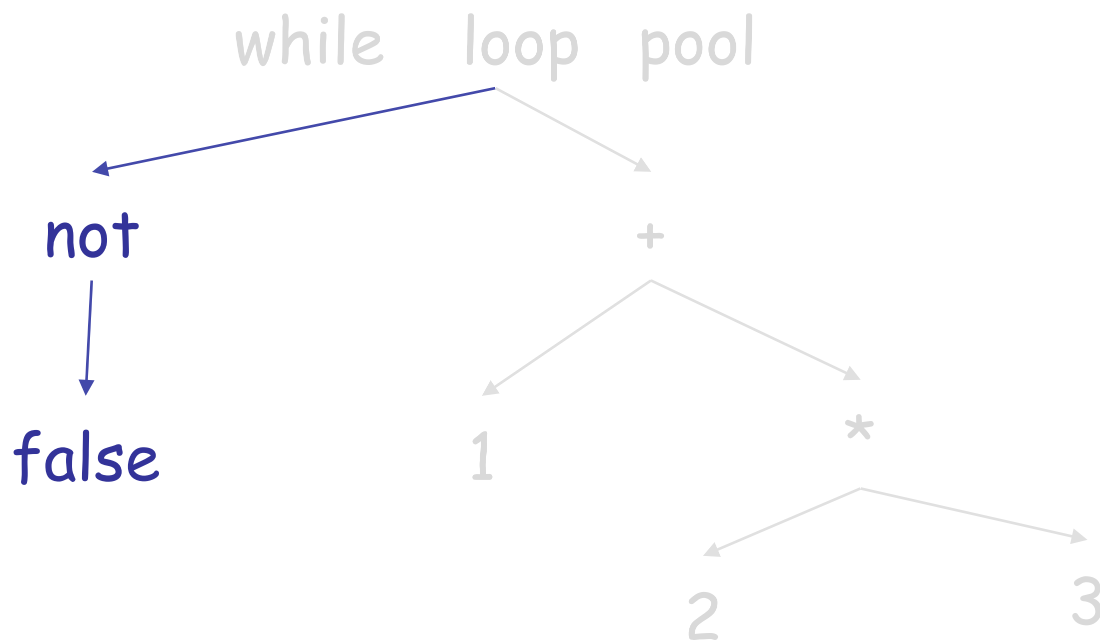
**Loop rule:**

$$\frac{\vdash e_1 : \text{Bool} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}}$$



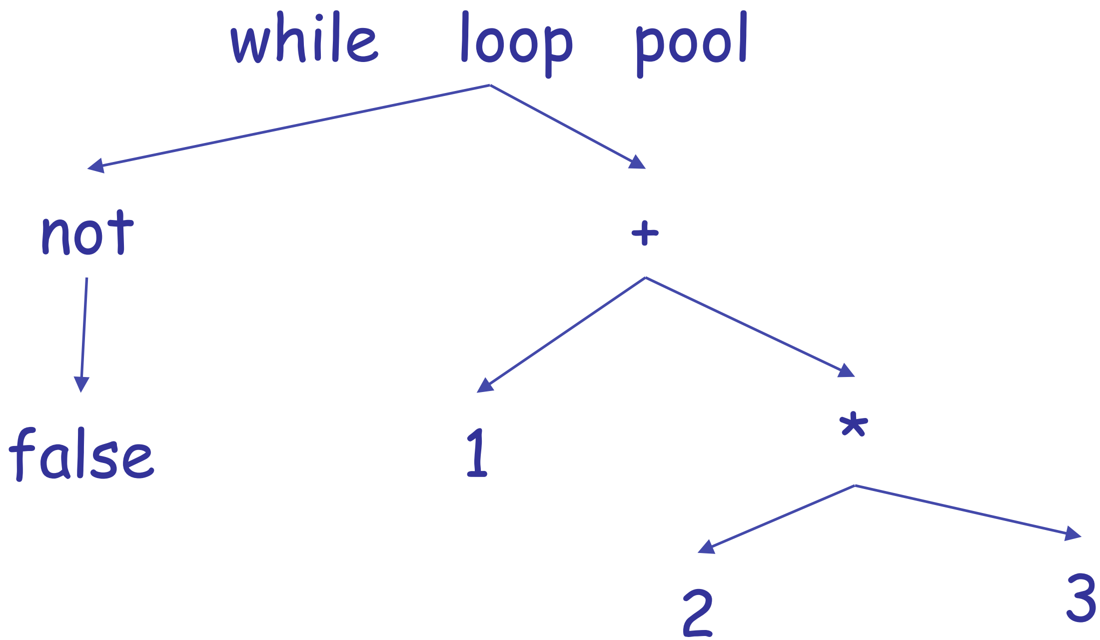
# Typing: Example

**while not false loop 1 + 2 * 3 pool**



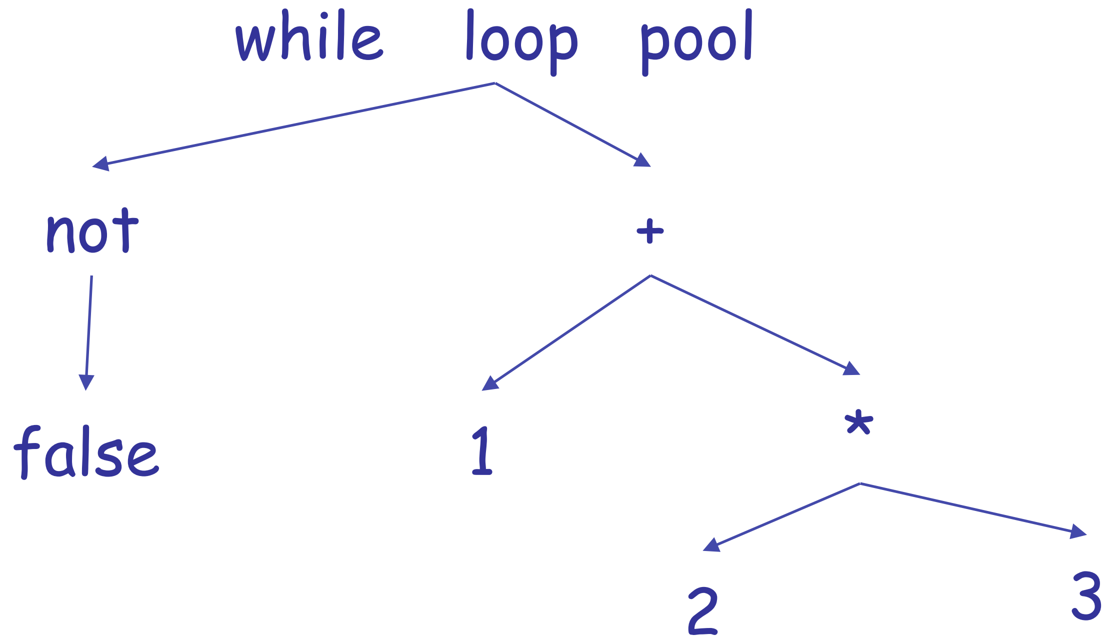
# Typing: Example

**while not false loop 1 + 2 * 3 pool**



# Typing: Example

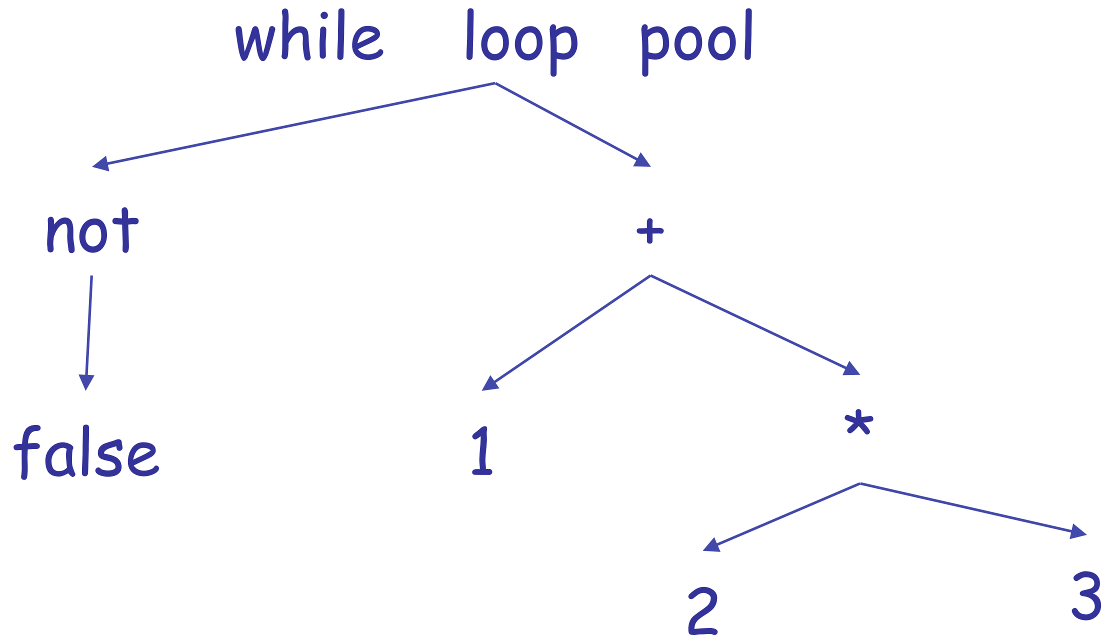
**while not false loop 1 + 2 * 3 pool**



# Typing: Example

**while not false loop 1 + 2 * 3 pool**

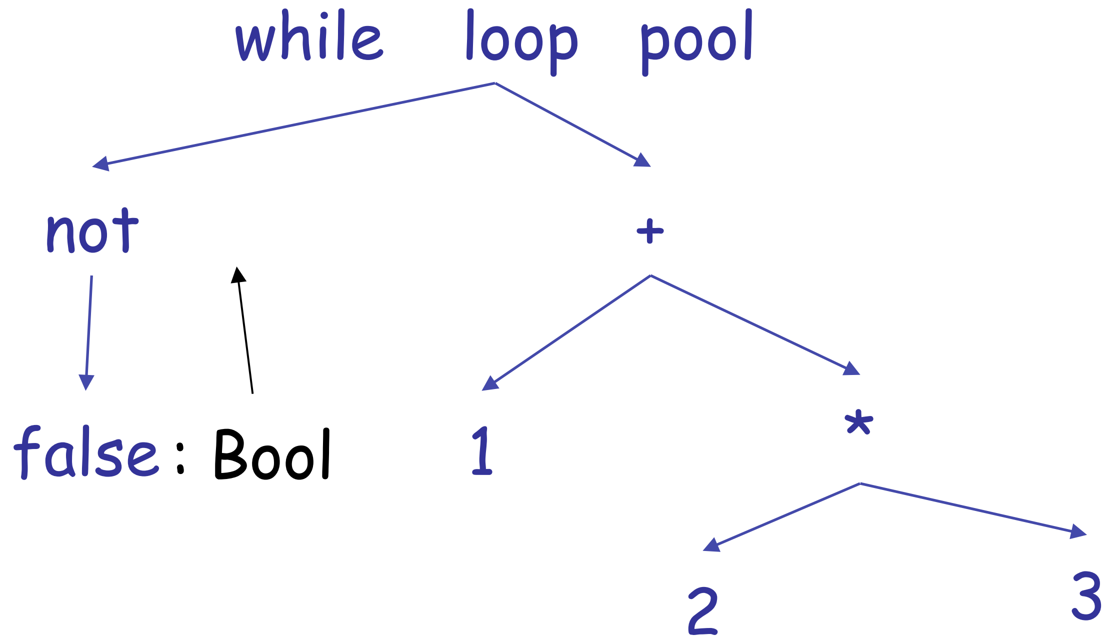
$$\frac{}{\vdash \text{false} : \text{Bool}}$$



# Typing: Example

**while not false loop 1 + 2 * 3 pool**

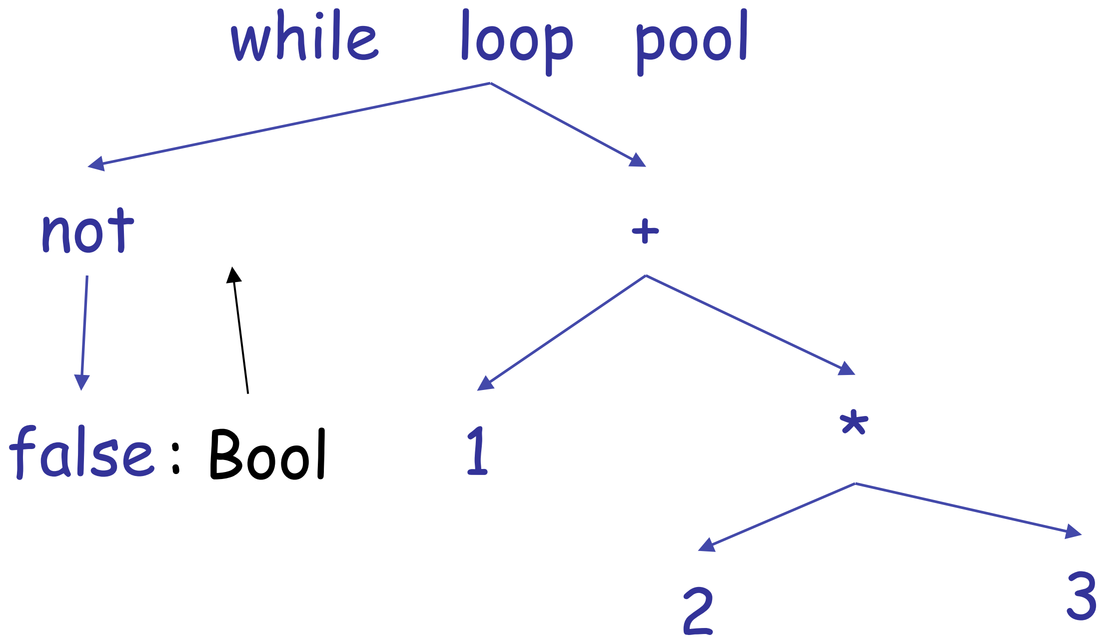
$$\frac{}{\vdash \text{false} : \text{Bool}}$$



# Typing: Example

**while not false loop 1 + 2 * 3 pool**

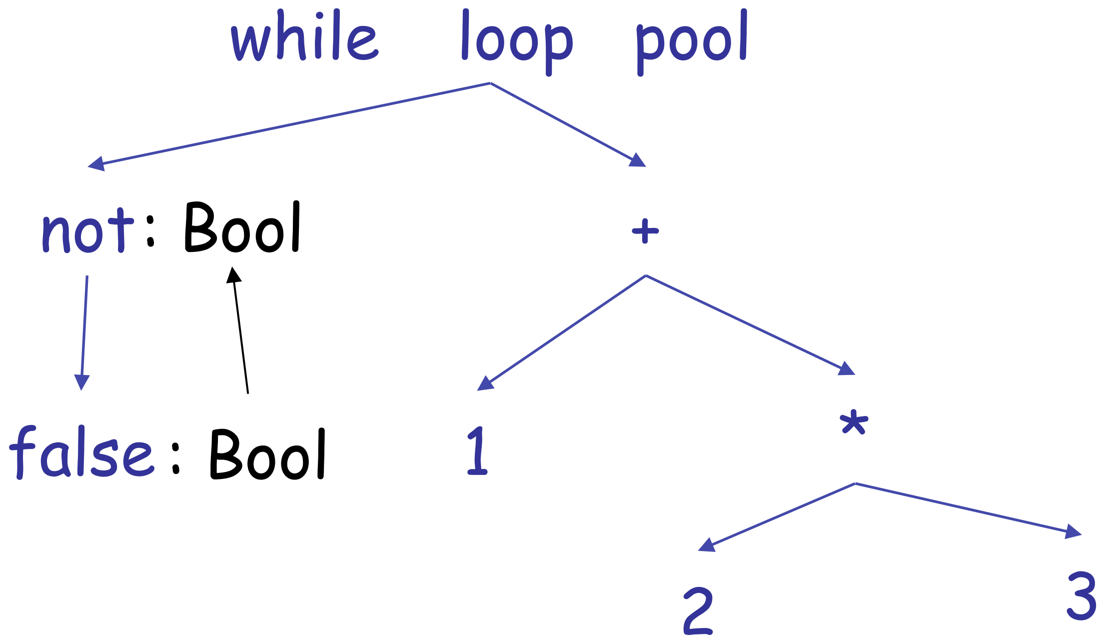
$$\frac{\vdash e : \text{Bool}}{\vdash \text{not } e : \text{Bool}}$$



# Typing: Example

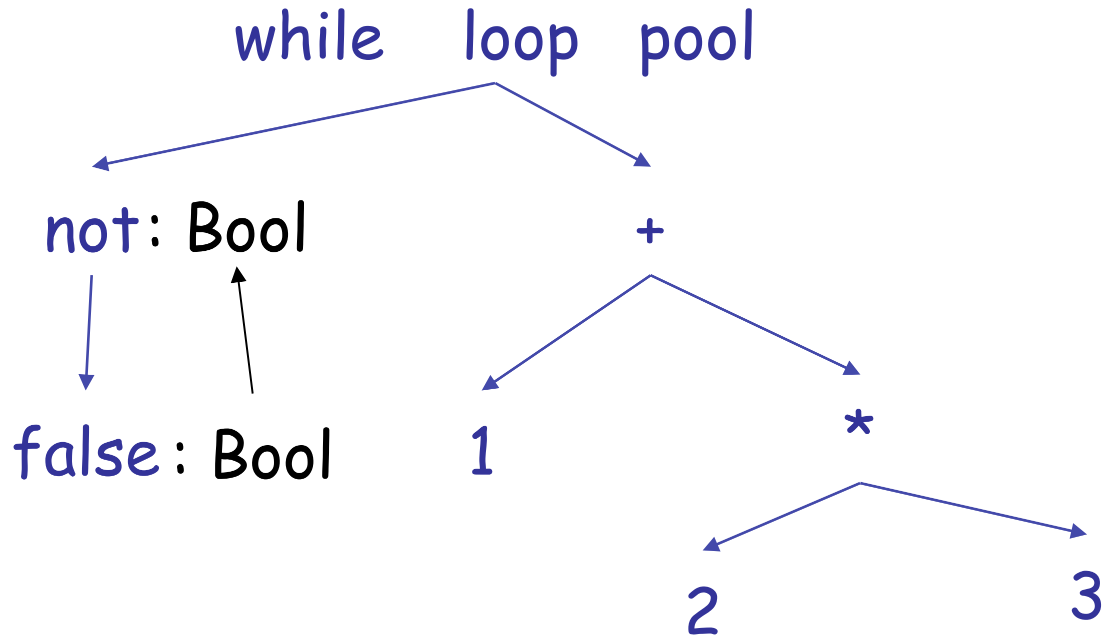
**while not false loop 1 + 2 * 3 pool**

$$\frac{\vdash e : \text{Bool}}{\vdash \text{not } e : \text{Bool}}$$



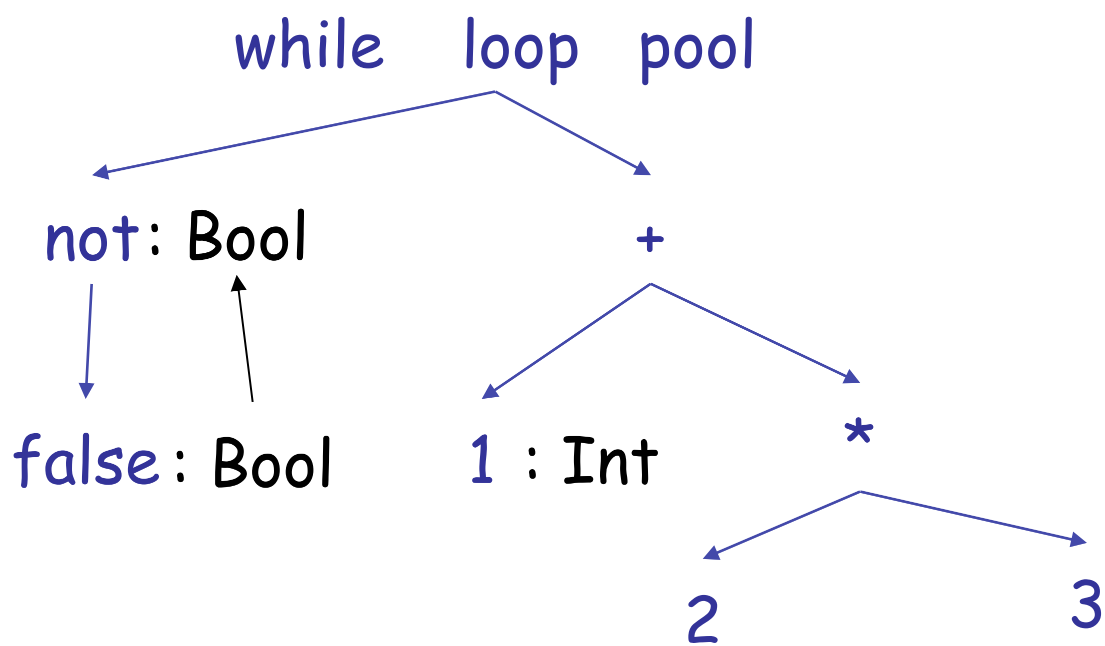
# Typing: Example

**while not false loop 1 + 2 * 3 pool**



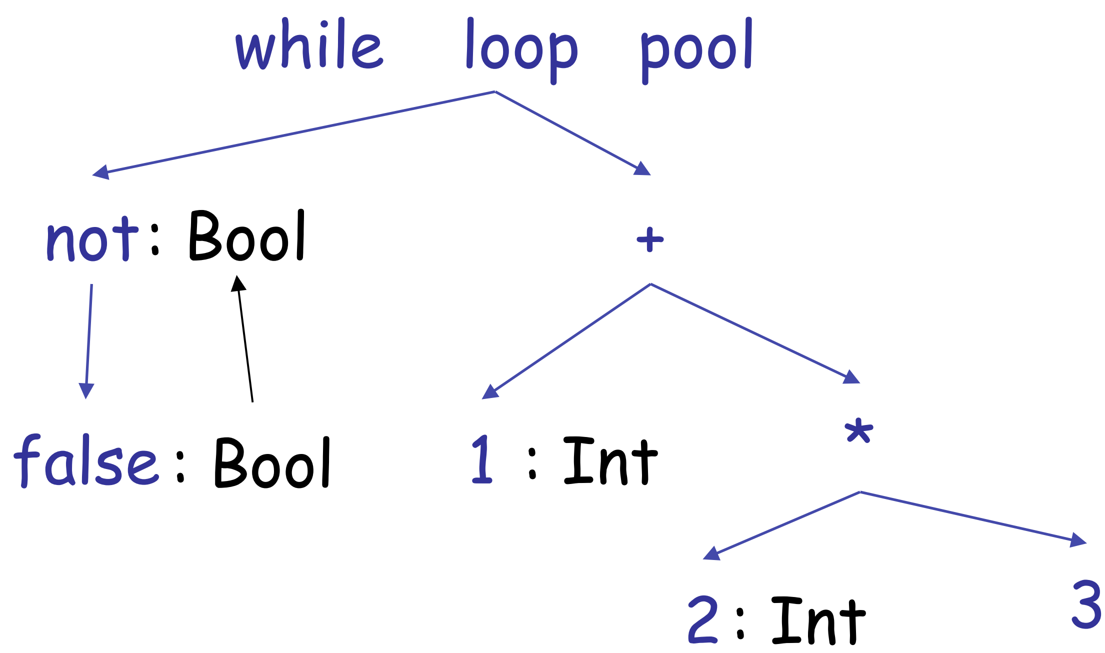
# Typing: Example

**while not false loop 1 + 2 * 3 pool**



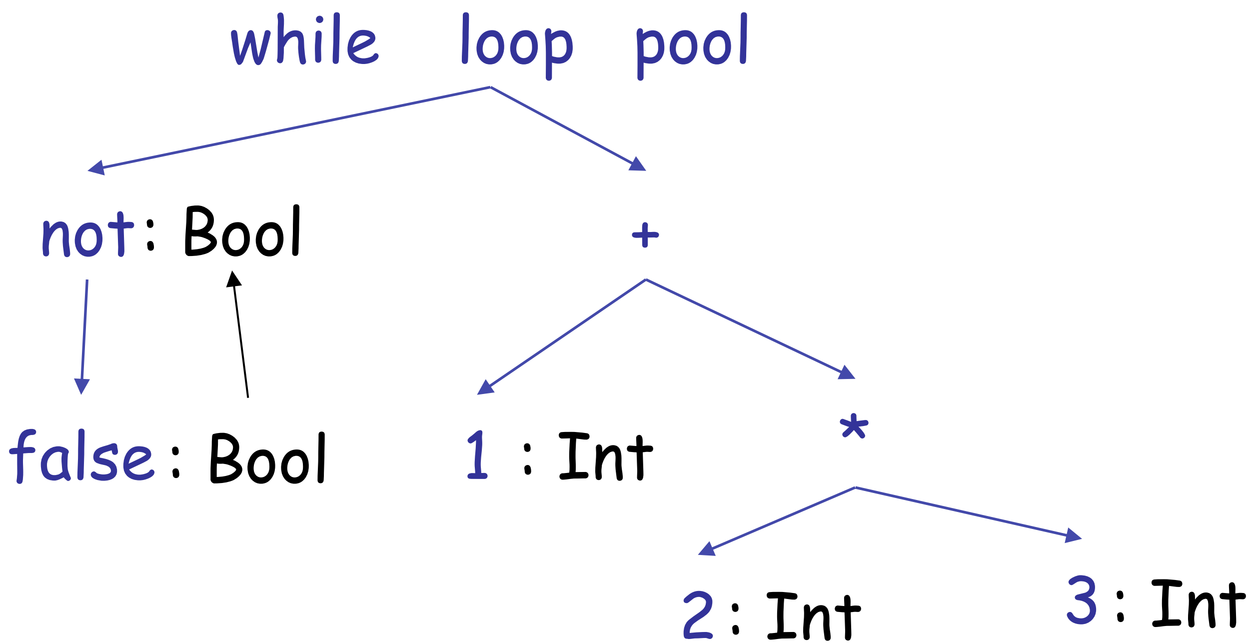
# Typing: Example

**while not false loop 1 + 2 * 3 pool**



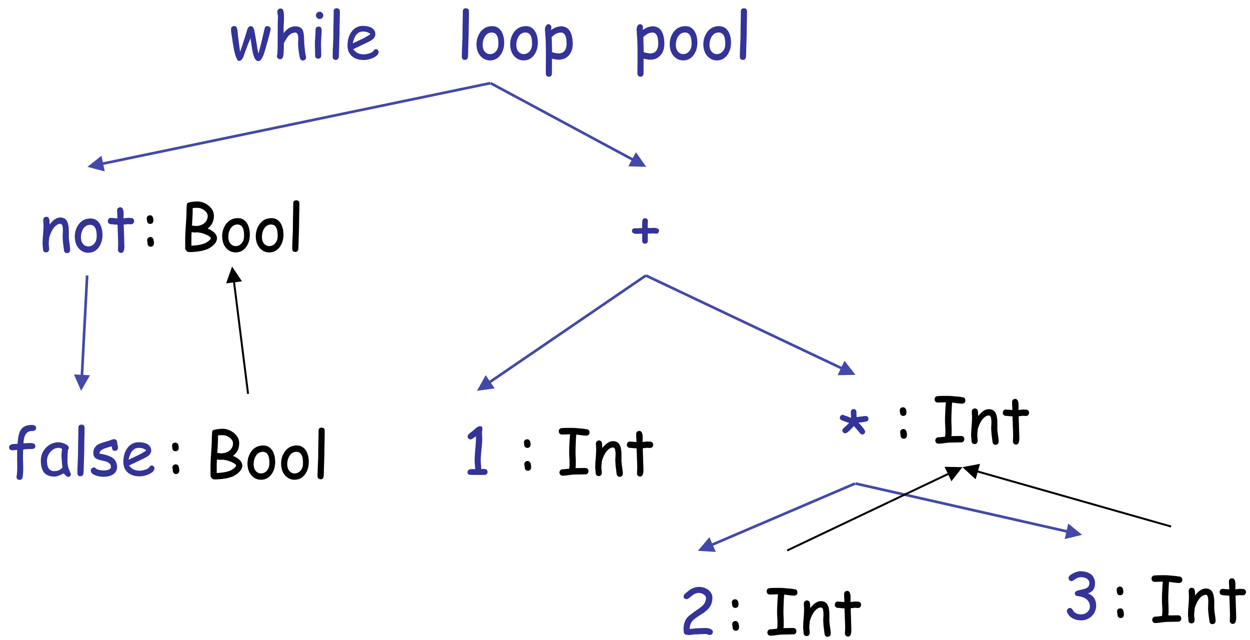
# Typing: Example

**while not false loop 1 + 2 * 3 pool**



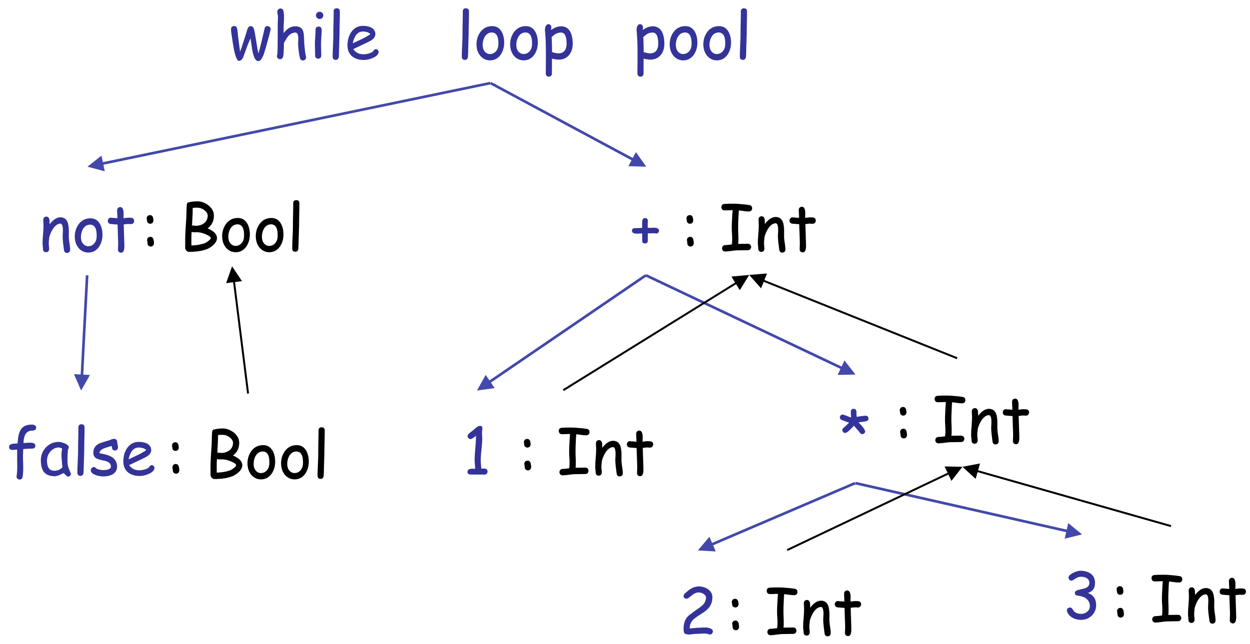
# Typing: Example

**while not false loop 1 + 2 * 3 pool**



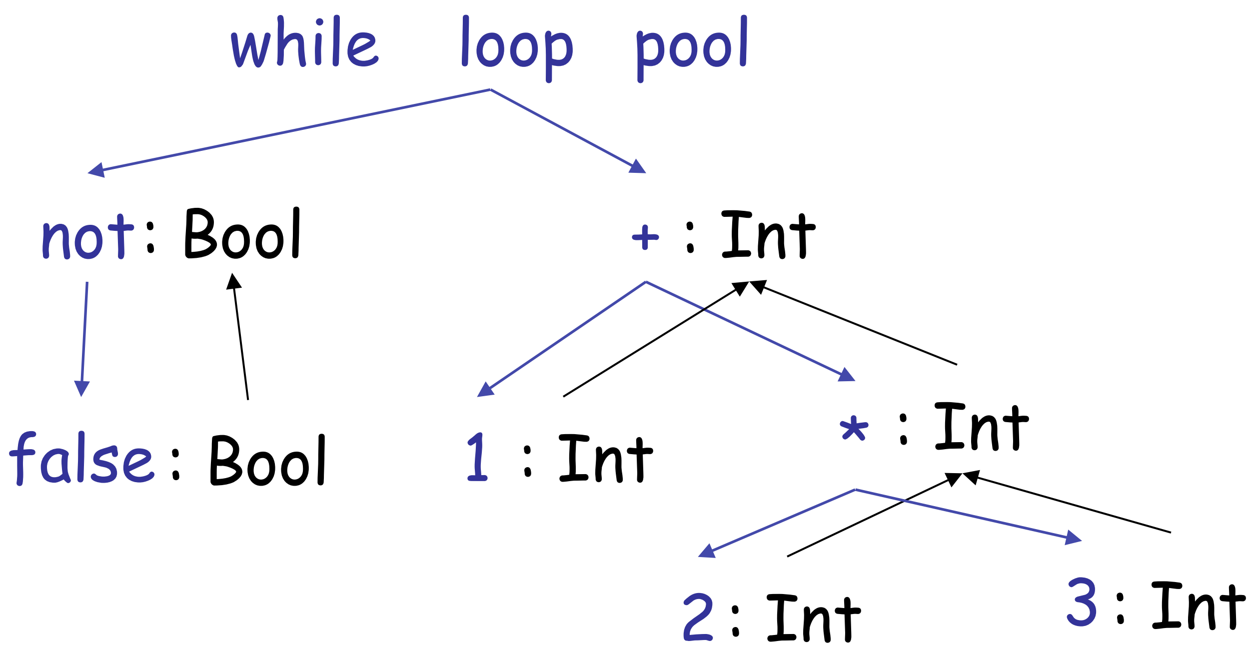
# Typing: Example

**while not false loop 1 + 2 * 3 pool**



# Typing: Example

**while not false loop 1 + 2 * 3 pool**

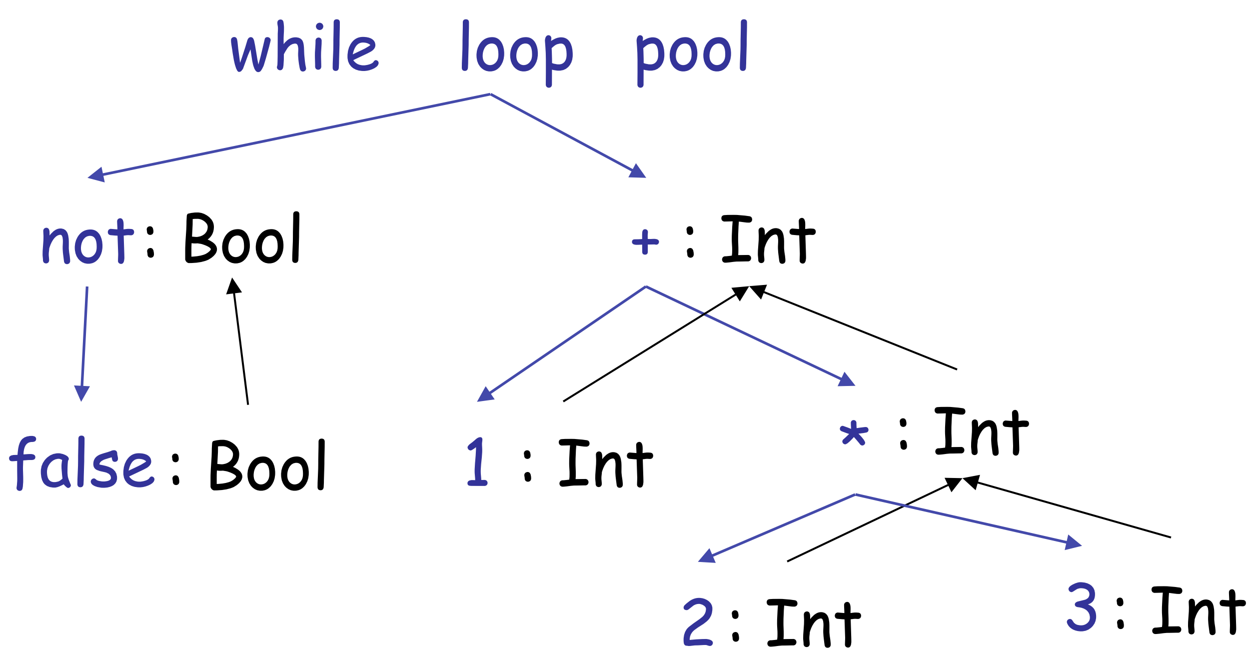


# Typing: Example

**while not false loop 1 + 2 * 3 pool**

**Loop rule:**

$$\frac{\vdash e_1 : \text{Bool} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}}$$

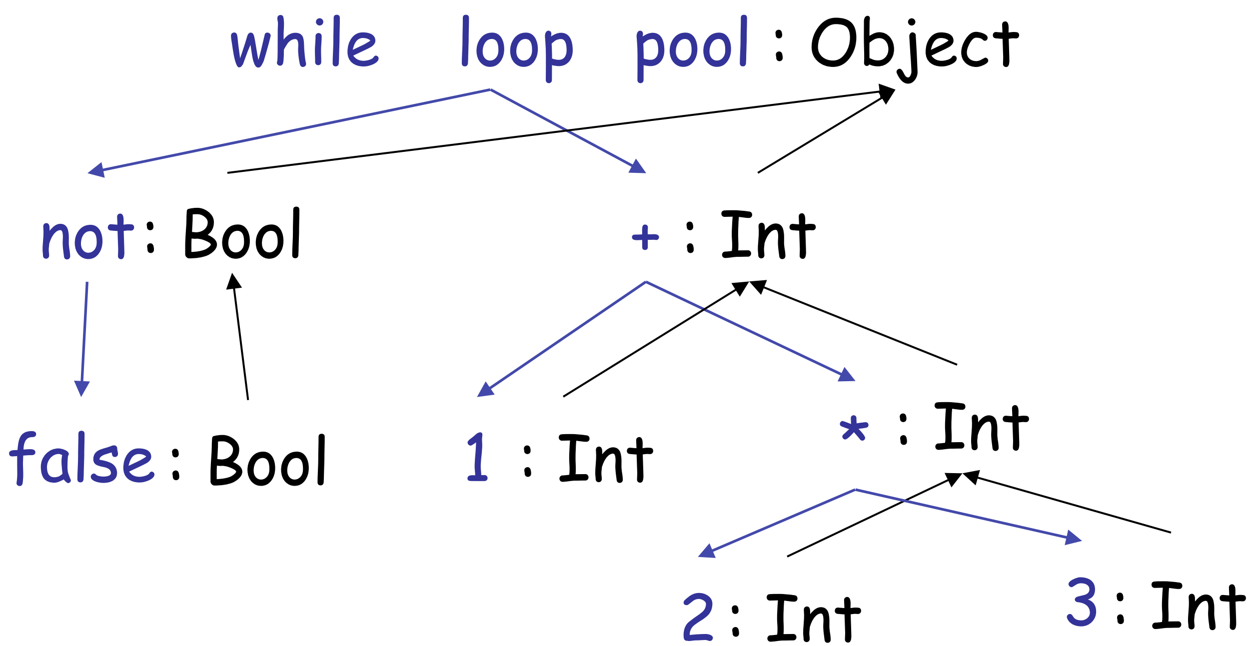


# Typing: Example

**while not false loop 1 + 2 * 3 pool**

**Loop rule:**

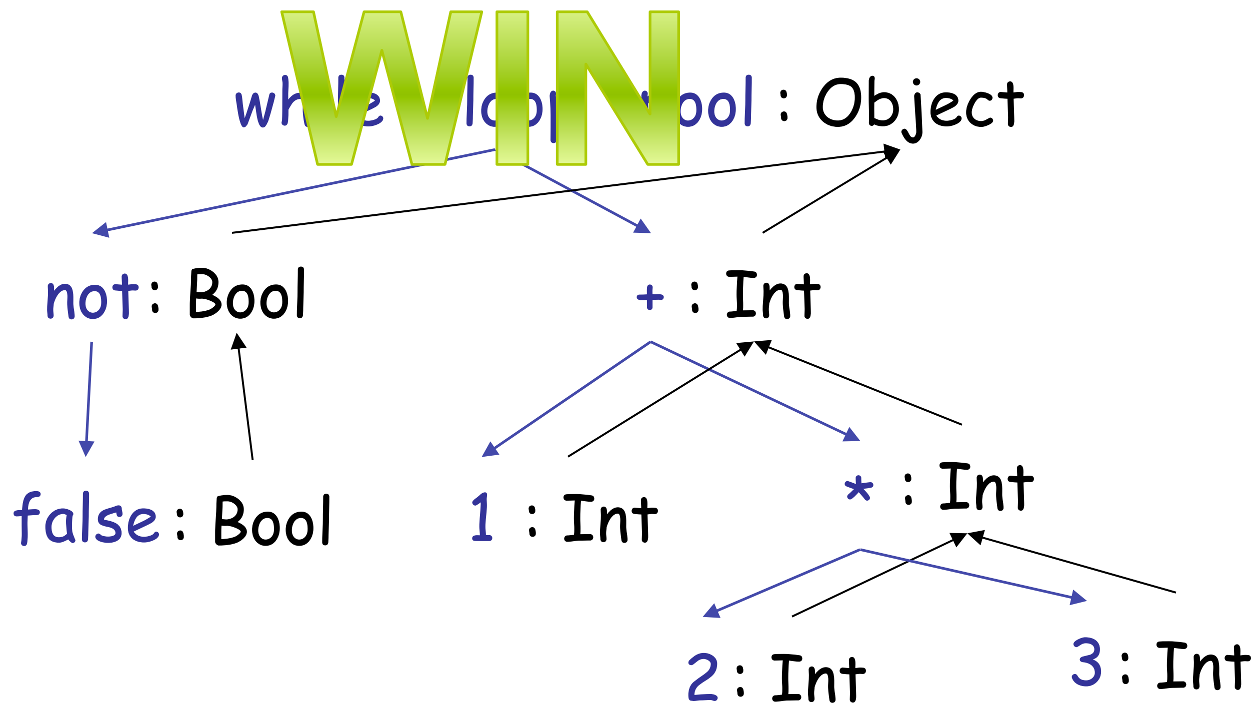
$$\frac{\vdash e_1 : \text{Bool} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}}$$



# Typing: Example

**while not false loop 1 + 2 * 3 pool**

**Loop rule:** $$\frac{\vdash e_1 : \text{Bool} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}}$$



# Typing Derivations

- This typing reasoning can be expressed as a tree:

$$\dfrac{\dfrac{\vdash \textbf{false} : \textbf{Bool}}{\vdash \textbf{not false} : \textbf{Bool}} \qquad \dfrac{\vdash \textbf{1} : \textbf{Int} \qquad \dfrac{\vdash \textbf{2} : \textbf{Int} \qquad \vdash \textbf{3} : \textbf{Int}}{\vdash \textbf{2 * 3} : \textbf{Int}}}{\vdash \textbf{1 + 2 * 3: Int}}}{\vdash \textbf{while not false loop 1 + 2 * 3 : Object}}$$

- The **root** of the tree is the whole expression (**while not false loop 1 + 2 * 3 pool**).
- Each node is an **instance** of a typing rule.
- **Leaves** are the rules with no hypotheses

# A Problem

- What is the type of a variable reference?

$$\frac{}{\vdash x : ?}\,[\text{Var}]$$ (x is an identifier)

- The local structural rule does ***not*** carry enough information to give x a type.

# A Problem

- What is the type of a variable reference?

$$\frac{}{\vdash x : ?}\ [\text{Var}] \quad (x \text{ is an identifier})$$

FAIL

- The local structural rule does ***not*** carry enough information to give x a type.

# Solution: add information to the rules!

- A **type environment** gives types for **free** variables. It is a mapping from Object_Identifiers to Types
  - A variable is **free** in an expression if the expression contains an occurrence of the variable that refers to a declaration *outside* the expression.
  - Examples:
    - in the expression “x”, the variable “x” is free
    - in “let x : Int in x + y” only “y” is free
    - in “x + let x : Int in x + y” both “x”, “y” are free

# Type Environments

- Let $O$ be a function from Object_Identifiers to Types
- The sentence $O \vdash e : T$ is read:
  - "Under the assumption that variables have the types given by $O$, it is provable that the expression $e$ has the type $T$."

# Modified Rules

- The type environment is added to the earlier rules:

$$\frac{}{O \vdash i : \text{Int}} \; [\text{Int}] \quad \text{(i is an integer)}$$

$$\frac{O \vdash e_1 : \text{Int} \qquad O \vdash e_2 : \text{Int}}{O \vdash e_1 + e_2 : \text{Int}} \; [\text{Add}]$$

# New Rules

- And we can write new rules:

$$\frac{}{O \vdash x : T} \text{[Var]} \quad (O(x) = T)$$

- Equivalently:

$$\frac{O(x) = T}{O \vdash x : T} \text{[Var]}$$

# Let

$$\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \textbf{let } x : T_0 \textbf{ in } e_1 : T_1} \quad \text{[Let-No-Init]}$$

- $O[T_0/x]$ means "$O$ modified to map $x$ to $T_0$ and behaving as $O$ on all other arguments":

$$O[T_0/x]\,(x) = T_0$$
$$O[T_0/x]\,(y) = O(y)$$

  - (You can write $O[x/T_0]$ on tests/assignments.)

# Let Example

- Consider the Cool expression

**let x : $T_0$ in (let y : $T_1$ in $E_{x, y}$) + (let x : $T_2$ in $F_{x, y}$)**

(where $E_{x, y}$ and $F_{x, y}$ are some Cool expression that contain occurrences of "x" and "y")

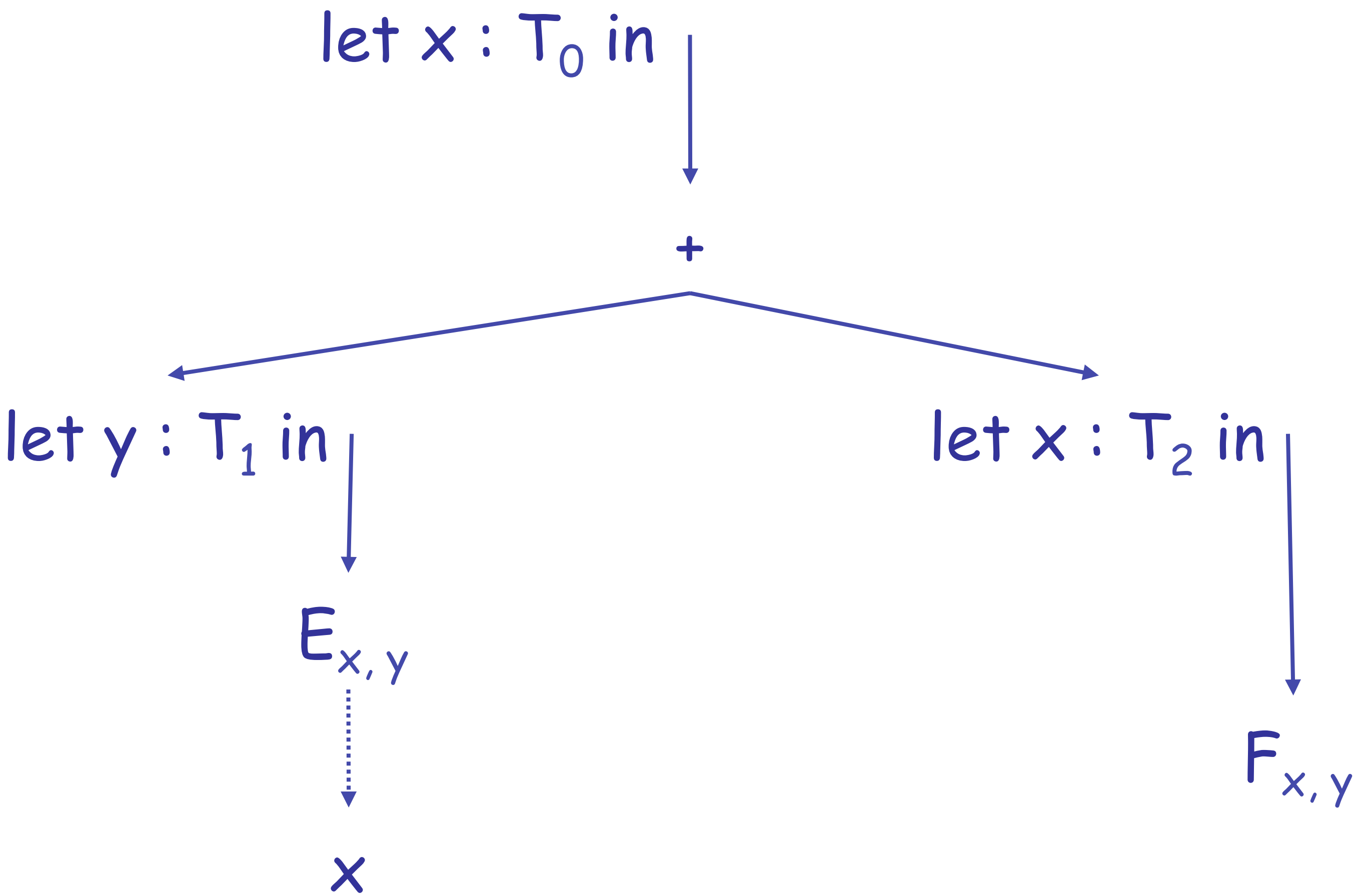
- Scope
  - of "y" is $E_{x, y}$
  - of outer "x" is $E_{x, y}$
  - of inner "x" is $F_{x, y}$
- This is captured precisely in the typing rule.

# Example of Typing "Let"

**let x : $T_0$ in (let y : $T_1$ in $E_{x,\ y}$) + (let x : $T_2$ in $F_{x,\ y}$)**

# Example of Typing "Let"

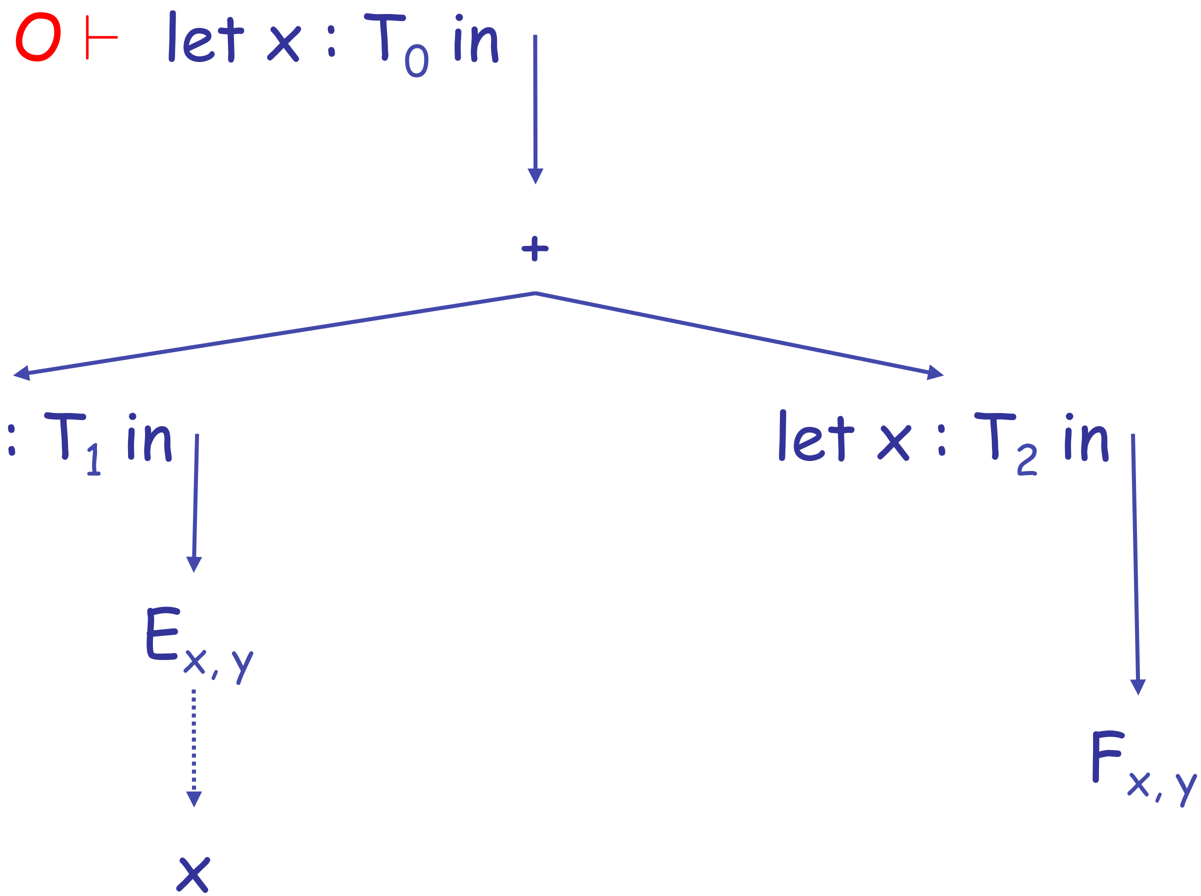
AST **let x : $T_0$ in (let y : $T_1$ in $E_{x, y}$) + (let x : $T_2$ in $F_{x, y}$)**



# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,\,y}$) + (let x : $T_2$ in $F_{x,\,y}$)**

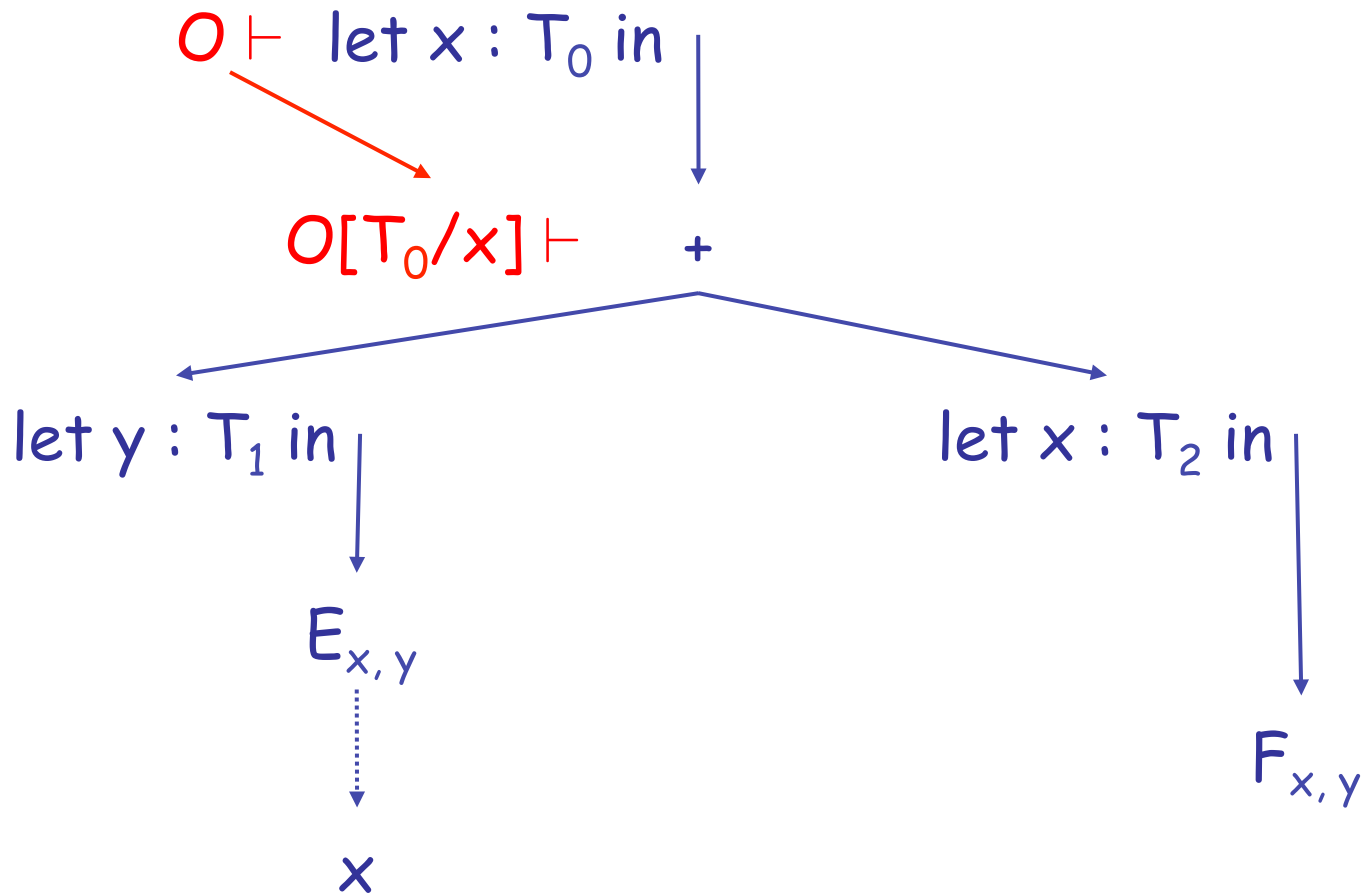
Type env.



# Example of Typing “Let”

AST **let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)**

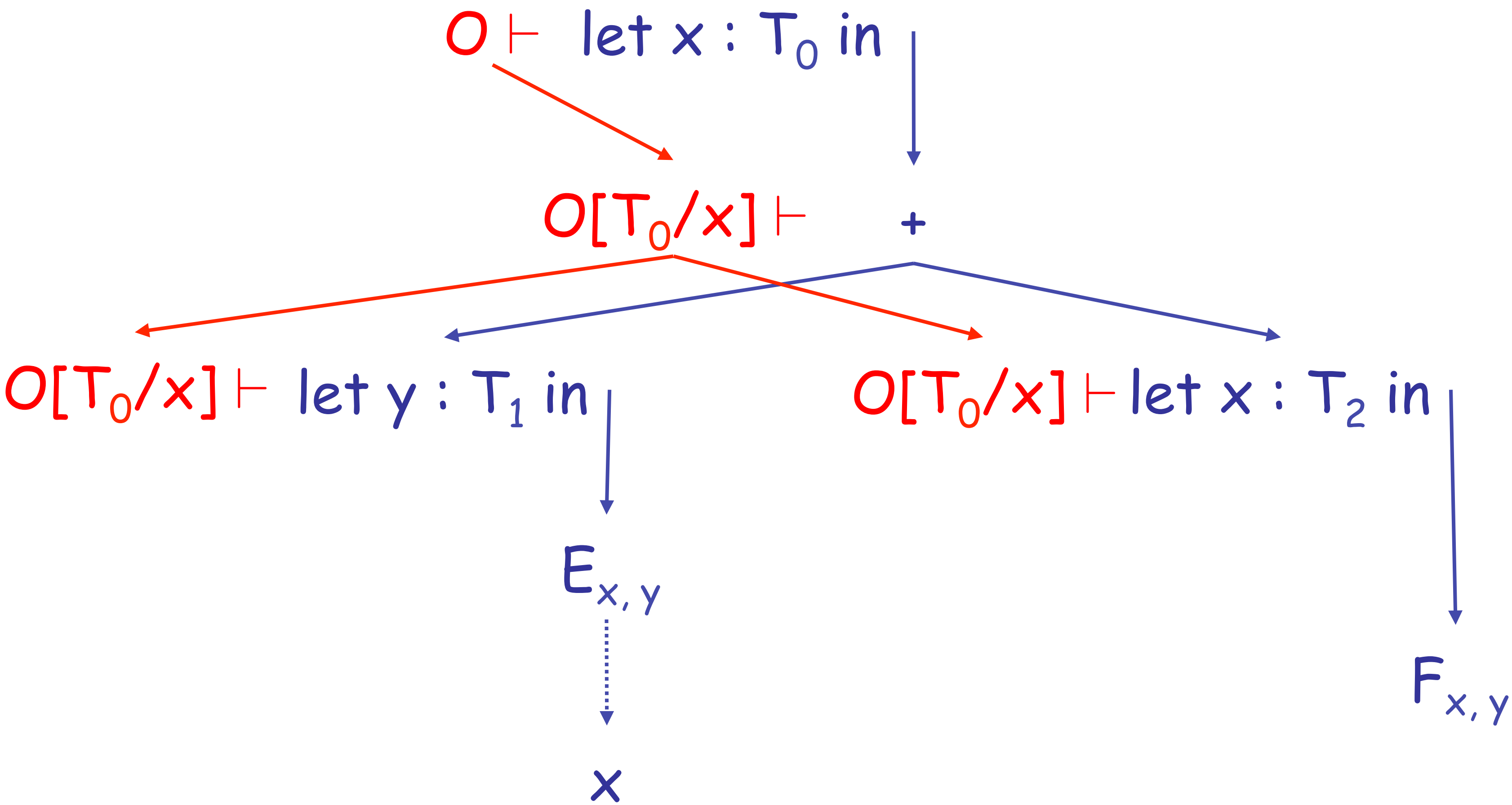
Type env.



# Example of Typing "Let"

AST **let x : $T_0$ in (let y : $T_1$ in $E_{x,\,y}$) + (let x : $T_2$ in $F_{x,\,y}$)**

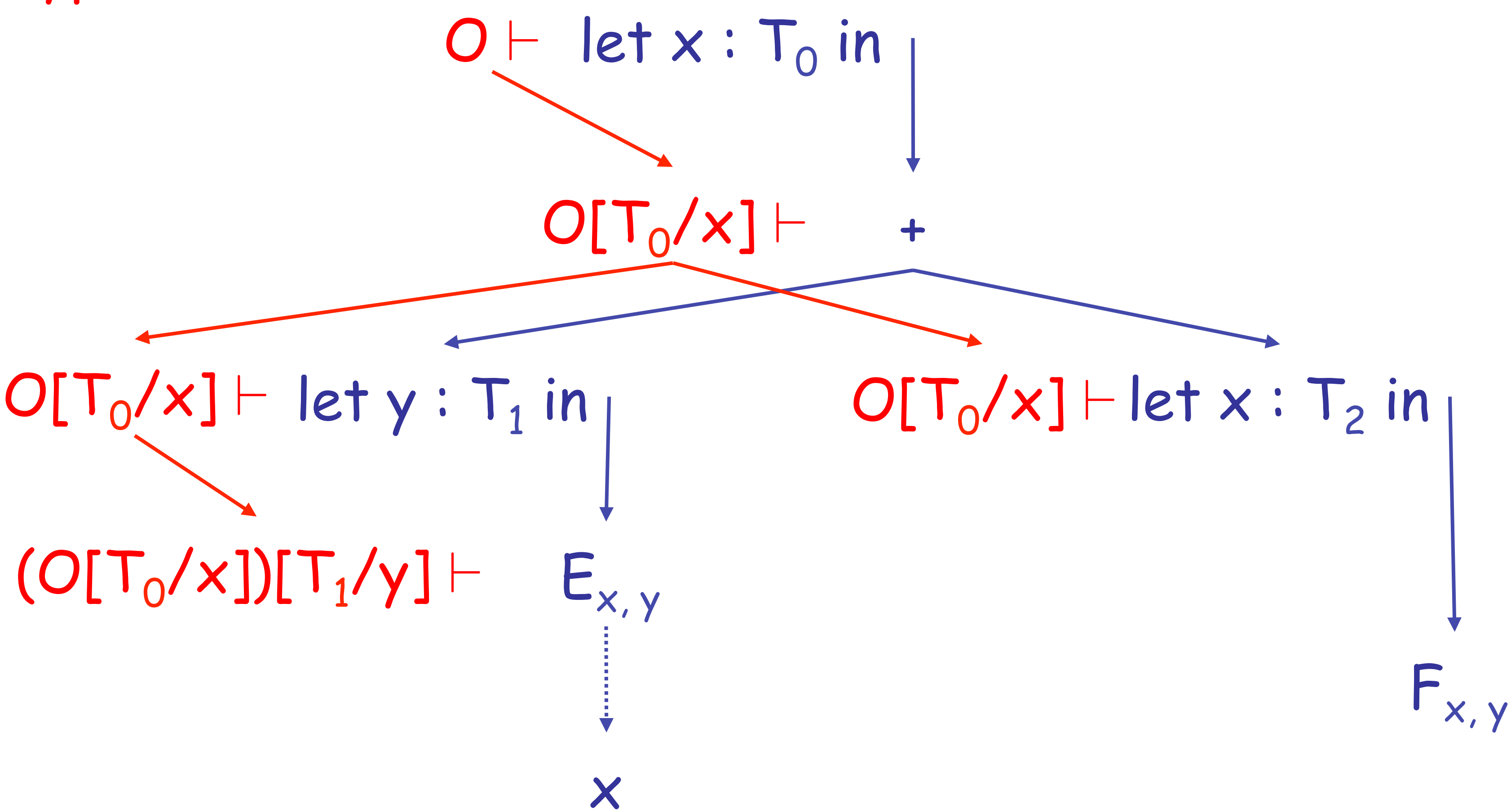
Type env.



# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,\,y}$) + (let x : $T_2$ in $F_{x,\,y}$)**

Type env.



# Example of Typing "Let"

AST **let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)**

Type env.

$O \vdash$ let x : $T_0$ in

$O[T_0/x] \vdash$ +

$O[T_0/x] \vdash$ let y : $T_1$ in

$O[T_0/x] \vdash$ let x : $T_2$ in

$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$

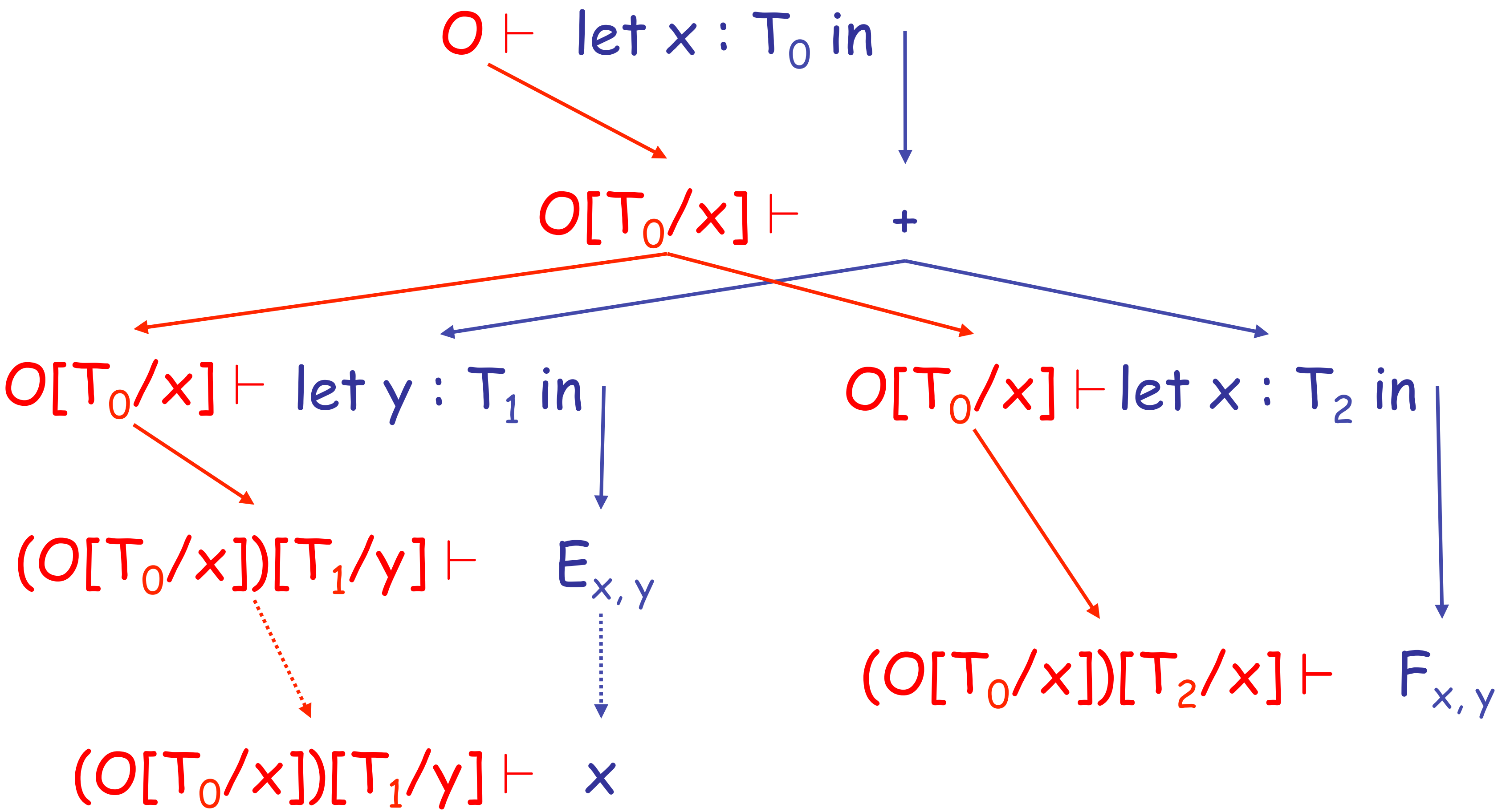
$F_{x,y}$

$(O[T_0/x])[T_1/y] \vdash$ x

# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)**

Type env.



# Example of Typing "Let"

**AST** **let x : $T_0$ in (let y : $T_1$ in $E_{x,\,y}$) + (let x : $T_2$ in $F_{x,\,y}$)**

Type env.

Types

$O \vdash$ let x : $T_0$ in

$O[T_0/x] \vdash$ +

$O[T_0/x] \vdash$ let y : $T_1$ in

$O[T_0/x] \vdash$ let x : $T_2$ in

$(O[T_0/x])[T_1/y] \vdash$ $E_{x,\,y}$

$(O[T_0/x])[T_2/x] \vdash$ $F_{x,\,y}$

$(O[T_0/x])[T_1/y] \vdash$ x : $T_0$

# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)**

Type env.

Types

$O \vdash$ let x : $T_0$ in

$O[T_0/x] \vdash$ +

$O[T_0/x] \vdash$ let y : $T_1$ in

$O[T_0/x] \vdash$ let x : $T_2$ in

$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$ : int

$(O[T_0/x])[T_2/x] \vdash$ $F_{x,y}$

$(O[T_0/x])[T_1/y] \vdash$ x : $T_0$

# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,\,y}$) + (let x : $T_2$ in $F_{x,\,y}$)**

Type env.

Types

$O \vdash$ let x : $T_0$ in

$O[T_0/x] \vdash$ +

$O[T_0/x] \vdash$ let y : $T_1$ in : int $\qquad O[T_0/x] \vdash$ let x : $T_2$ in

$(O[T_0/x])[T_1/y] \vdash$ $E_{x,\,y}$ : int

$(O[T_0/x])[T_2/x] \vdash$ $F_{x,\,y}$

$(O[T_0/x])[T_1/y] \vdash$ x : $T_0$

# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)**

Type env.

Types

$O \vdash$ let x : $T_0$ in

$O[T_0/x] \vdash$ +

$O[T_0/x] \vdash$ let y : $T_1$ in : int

$O[T_0/x] \vdash$ let x : $T_2$ in

$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$ : int

$(O[T_0/x])[T_2/x] \vdash$ $F_{x,y}$ : int

$(O[T_0/x])[T_1/y] \vdash$ x : $T_0$

# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)**

Type env.

Types

$O \vdash$ let x : $T_0$ in

$O[T_0/x] \vdash$ +

$O[T_0/x] \vdash$ let y : $T_1$ in : int

$O[T_0/x] \vdash$ let x : $T_2$ in : int

$(O[T_0/x])[T_1/y] \vdash E_{x,y}$ : int

$(O[T_0/x])[T_2/x] \vdash F_{x,y}$ : int

$(O[T_0/x])[T_1/y] \vdash x$ : $T_0$

# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,\,y}$) + (let x : $T_2$ in $F_{x,\,y}$)**

Type env.

Types

$O \vdash$ let x : $T_0$ in

$O[T_0/x] \vdash \quad + \quad : int$

$O[T_0/x] \vdash$ let y : $T_1$ in $\quad$ : int $\qquad$ $O[T_0/x] \vdash$ let x : $T_2$ in $\quad$ : int

$(O[T_0/x])[T_1/y] \vdash \quad E_{x,\,y} \quad : int$

$(O[T_0/x])[T_2/x] \vdash \quad F_{x,\,y} \quad : int$

$(O[T_0/x])[T_1/y] \vdash \quad x \quad : T_0$

# Notes

- The type environment gives types to the free identifiers *in the current scope*
- The **type environment** is passed down the AST from the root towards the leaves
- **Types** are computed up the AST from the leaves towards the root

# Example of Typing "Let"

AST $\quad$ **let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)**

Type env.

Types

$O \vdash$ let x : $T_0$ in

$O[T_0/x] \vdash \quad + \quad : int$

$O[T_0/x] \vdash$ let y : $T_1$ in $\quad : int \qquad O[T_0/x] \vdash$ let x : $T_2$ in $\quad : int$

$(O[T_0/x])[T_1/y] \vdash \quad E_{x,y} \quad : int$

$(O[T_0/x])[T_2/x] \vdash \quad F_{x,y} \quad : int$

$(O[T_0/x])[T_1/y] \vdash \quad x \quad : T_0$

# Q: Movies (362 / 842)

- In this 1992 comedy Dana Carvey and Mike Myers reprise a **Saturday Night Live** skit, sing **Bohemian Rhapsody** and say of a guitar: *"Oh yes, it will be mine."*

# Q: General (455 / 842)

- This numerical technique for finding solutions to boundary-value problems was initially developed for use in structural analysis in the 1940's. The subject is represented by a model consisting of a number of linked simplified representations of discrete regions. It is often used to determine stress and displacement in mechanical systems.

# Q: Movies (377 / 842)

- Identify the subject or the speaker in 2 of the following 3 **Star Wars** quotes.
  - "Aren't you a little short to be a stormtrooper?"
  - "I felt a great disturbance in the Force ... as if millions of voices suddenly cried out in terror and were suddenly silenced."
  - "I recognized your foul stench when I was brought on board."

# Let with Initialization

- Now consider let with initialization:

$$\frac{O \vdash e_0 : T_0 \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \text{[Let-Init]}$$

- This rule is weak. Why?

# Let with Initialization

- Consider the example:

  class C inherits P { … }

  …

  let x : P ← new C in …

  …

- The previous let rule *does not allow this code*
  - The rule is **too weak** or **incomplete**.

# Subtyping

- Define a relation $X \leq Y$ on classes to say that:
  - An object of type X could be used when one of type Y is acceptable, or equivalently, *X conforms with Y*
  - In Cool, this means that X is a **subclass** of Y
- Define a relation $\leq$ on classes such that:

  $X \leq X$

  $X \leq Y$ if X inherits from Y

  $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

# Let With Initialization (Better)

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ T \leq T_0 \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \quad \text{[Let-Init]}$$

- Both rules for let are sound
- But more programs type check with this new rule (it is more complete).

# Dynamic And Static Types

- The **dynamic type** of an object is the class C that is used in the "new C" expression that creates the object
  - A **run-time** notion.
  - Even languages that are not statically typed have the notion of dynamic type.
- The **static type** of an expression is a notation that captures all possible dynamic types the expression could take.
  - A **compile-time** notion

# Dynamic and Static Types. (Cont.)

- In early type systems, the set of static types correspond **directly** with the dynamic types.
- Soundness theorem: for all expressions E

dynamic_type(E) = static_type(E)

(in **all** executions, E evaluates to values of the type inferred by the compiler)

- This gets more complicated in advanced type systems (e.g., Java, Cool)

# Type System Tug-of-War

- There is a tension between
  - Flexible rules that do not constrain programming
  - Restrictive rules that ensure safety of execution

# Static Type System Expressiveness

- A **static** type system enables a compiler to detect many common programming errors
- The cost is that some correct programs are disallowed
  - Some argue for dynamic type checking instead.
  - …others argue for more expressive static type checking.
- But more expressive type systems are also more complex

# Dynamic and Static Types in COOL

- A variable of static type A can hold values of static type B, if B $\leq$ A

```
class A { … }
class B inherits A {…}
class Main {
   A x ← new A;
   …
   x ← new B;
   …
}
```

**x has static type A**

**Here, x's value has dynamic type A**

**Here, x's value has dynamic type B**

# Dynamic and Static Types

- **Soundness theorem** for the Cool type system:
  $\forall$ E. dynamic_type(E) $\leq$ static_type(E)
- Why is this Ok?
  - For E, compiler uses static_type(E)
  - All operations that can be used on an object of type C can also be used on an object of type C' $\leq$ C
    - Examples: fetching the value of an attribute, or invoking a method on the object
  - Subclasses can ***only add*** attributes or methods
  - Methods can be redefined, but with the same types!

# Subtyping Example

- Consider the following Cool class definitions

  Class A { a() : int { 0 }; }

  Class B inherits A { b() : int { 1 }; }

- An instance of B has methods “a” and “b”
- An instance of A has method “a”
  - A type error occurs if we try to invoke method “b” on an instance of A

# Example of Wrong Let Rule (1)

- Now consider a hypothetical wrong let rule:

$$\frac{O \vdash e_0 : T \qquad T \leq T_0 \qquad O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?

# Example of Wrong Let Rule (1)

- Now consider a hypothetical wrong let rule:

$$\frac{O \vdash e_0 : T \qquad T \leq T_0 \qquad O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?
- The following good program does *not* typecheck

$$\text{let } x : \text{Int} \leftarrow 0 \text{ in } x + 1$$

- Why?

# Example of Wrong Let Rule (2)

- Now consider a hypothetical wrong let rule:

$$\frac{O \vdash e_0 : T \quad T_0 \leq T \quad O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?

# Example of Wrong Let Rule (2)

- Now consider another hypothetical wrong let rule:

$$\frac{O \vdash e_0 : T \quad T_0 \leq T \quad O[T_0/x] \vdash e_1 : T_1}{O \vdash \textbf{let } x : T_0 \leftarrow e_0 \textbf{ in } e_1 : T_1}$$

- How is it different from the correct rule?
- The following *bad* program is well typed

**let x : B ← new A in x.b()**

- Why is this program bad?

# Example of Wrong Let Rule (3)

- Now consider a third hypothetical wrong let rule:

$$\frac{O \vdash e_0 : T \quad T \leq T_0 \quad O[T/x] \vdash e_1 : T_1}{O \vdash \textbf{let } x : T_0 \leftarrow e_0 \textbf{ in } e_1 : T_1}$$

- How is it different from the correct rule?

# Example of Wrong Let Rule (3)

- Now consider a third hypothetical wrong let rule:

$$\frac{O \vdash e_0 : T \quad T \leq T_0 \quad O[T/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?
- The following good program is *not* well typed

**let x : A ← new B in {… x ← new A; x.a(); }**

- Why is this program not well typed?

# Upshot: Typing Rule Notation

- The typing rules use **very concise** notation
- They are very carefully constructed
- Virtually any change in a rule either:
  - Makes the type system **unsound** (i.e., bad programs are accepted as well typed)
  - Or, makes the type system less usable or **incomplete** (i.e. good programs are rejected)
- But some good programs will always be rejected, because the notion of a good program is **undecidable**

# Assignment

- More uses of subtyping:

$$\frac{O(id) = T_0 \quad O \vdash e_1 : T_1 \quad T_1 \leq T_0}{O \vdash id \leftarrow e_1 : T_1} \; [Assign]$$

# Initialized Attributes

- Let $O_C(x) = T$ for all attributes $x{:}T$ in class $C$
  - $O_C$ represents the class-wide scope
- Attribute initialization is similar to let, except for the scope of names

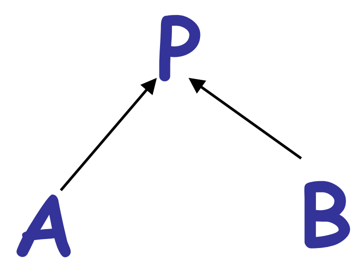
$$\frac{\begin{array}{c} O_C(id) = T_0 \\ O_C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C \vdash id : T_0 \leftarrow e_1 \ ;} \quad \text{[Attr-Init]}$$

# If-Then-Else

- Consider: **if $e_0$ then $e_1$ else $e_2$ fi**
- The result can be either $e_1$ or $e_2$; the dynamic type is therefore either $e_1$'s or $e_2$'s type.
- The best we can do statically is the **smallest supertype** larger than both the types of $e_1$ and $e_2$

# If-Then-Else example

- Consider the class hierarchy




- … and the expression

**if … then new A else new B fi**

- Its type should allow for the dynamic type to be both A or B
  - Smallest supertype is P

# Least Upper Bounds

- Define: **lub(X,Y)** to be the **least upper bound** of X and Y. lub(X,Y) is Z if
  - $X \le Z \wedge Y \le Z$

Z is an upper bound

- $X \le Z' \wedge Y \le Z' \Rightarrow Z \le Z'$

Z is least among upper bounds

- In Cool, the least upper bound of two types is their **least common ancestor** in the **inheritance tree**

# If-Then-Else Revisited

$$\frac{O \vdash e_0 : \text{Bool} \quad O \vdash e_1 : T_1 \quad O \vdash e_2 : T_2}{O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \textbf{lub}(T_1, T_2)}$$

[If-Then-Else]

# Case

- The rule for case expressions takes a lub over all branches

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O[T_1/x_1] \vdash e_1 : T_1' \\ \ldots \\ O[T_n/x_n] \vdash e_n : T_n' \end{array}}{\begin{array}{c} O \vdash \texttt{case } e_0 \texttt{ of } x_1{:}T_1 \Rightarrow e_1; \\ \ldots; x_n : T_n \Rightarrow e_n; \texttt{ esac} : lub(T_1', \ldots, T_n') \end{array}} \quad [Case]$$

# Next Time

- Type checking method dispatch
- Type checking with SELF_TYPE in COOL

# Homework

- Get started on PA4
  - **Checkpoint due March 19**
- Before Next Class: Read Chapter 7.2