Symbol Table (Γ) Contents

- Map identifiers to the symbol with relevant information about the identifier
- All information is derived from syntax tree - symbol table is for efficiency
  - in old one-pass compilers there was only symbol table, no syntax tree
  - in modern compiler: we could always go through entire tree, but symbol table can give faster and easier access to the part of syntax tree, or some additional information
- Goal: efficiently supporting phases of compiler
- In the name analysis phase:
  - finding which identifier refers to which definition
  - we store *definitions*
- What kinds of things can we define? What do we need to know for each ID?

variables (globals, fields, parameters, locals):

- need to know types, positions - for error messages
- later: memory layout. To compile x.f = y into memcopy(addr_y, addr_x+6, 4)
  - e.g. 3rd field in an object should be stored at offset e.g. +6 from the address of the object
  - the size of data stored in x.f is 4 bytes
- sometimes more information explicit: whether variable local or global

methods, functions, classes: recursively have with their own symbol tables

Different Points, Different Γ

```
class World {
  int sum;
  void add(int foo) {
    sum = sum + foo;
  }
  void sub(int bar) {
    sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0 [foo := int]$

Γ_0

$\Gamma_1 = \Gamma_0 [bar := int]$

Imperative Way: Push and Pop

```
class World {
  int sum;
  void add(int foo) {
     sum = sum + foo;
  }
  void sub(int bar) {
     sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0 [foo := int]$

change table, record change

Γ_0 revert changes from table

$\Gamma_1 = \Gamma_0 [bar := int]$

change table, record change

revert changes from table

Imperative Symbol Table

- Hash table, mutable Map[ID,Symbol]
- Example:
  - hash function into array
  - array has linked list storing (ID,Symbol) pairs
- Undo stack: to enable entering and leaving scope
- Entering new scope (function,block):
  - add beginning-of-scope marker to undo stack
- Adding nested declaration (ID,sym)
  - lookup old value symOld, push old value to undo stack
  - insert (ID,sym) into table
- Leaving the scope
  - go through undo stack until the marker, restore old values

Functional: Keep Old Version

```
class World {
  int sum;
  void add(int foo) {
     sum = sum + foo;
  }
  void sub(int bar) {
     sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0[foo := int]$

create new Γ_1, keep old Γ_0

Γ_0

$\Gamma_2 = \Gamma_0[bar := int]$

create new Γ_1, keep old Γ_0

Functional Symbol Table Implemented

- Typical: Immutable Balanced Search Trees

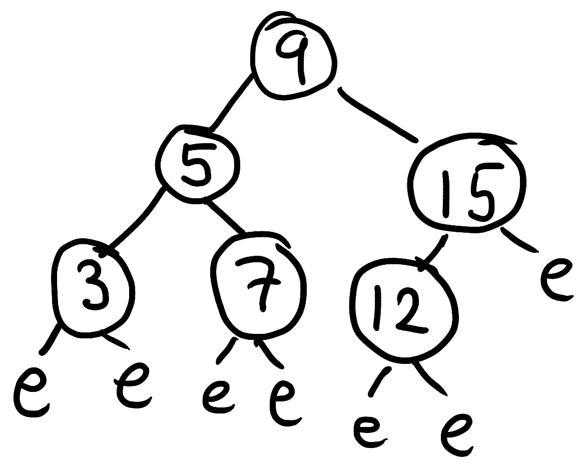
Simplified. In practice, BST[A], store Int key and value A

```
sealed abstract class BST
case class Empty() extends BST
case class Node(left: BST, value: Int, right: BST) extends BST
```

- Updating returns new map, keeping old one
  - lookup and update both *log(n)*
  - update creates new path (copy *log(n)* nodes, share rest!)
  - memory usage acceptable

Lookup

```
def contains(key: Int, t : BST): Boolean = t match {
    case Empty() => false
    case Node(left,v,right) => {
        if (key == v) true
        else if (key < v) contains(key, left)
        else contains(key, right)
    }
}
```

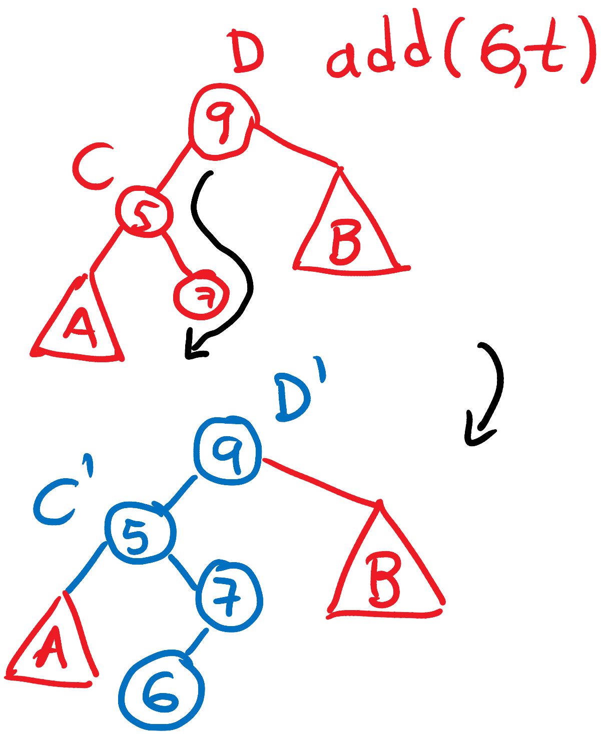


Running time bounded by tree height.

contains(6,t) ?

Insertion

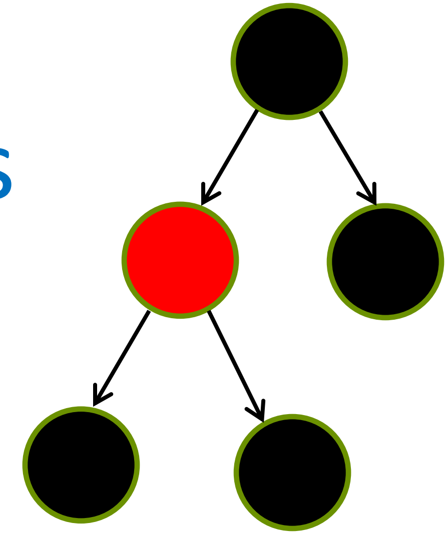
```
def add(x : Int, t : BST) : Node = t match {
  case Empty() => Node(Empty(),x,Empty())
  case t @ Node(left,v,right) => {
    if (x < v) Node(add(x, left), v, right)
    else if (x==v) t
    else Node(left, v, add(x, right))
  }
}
```

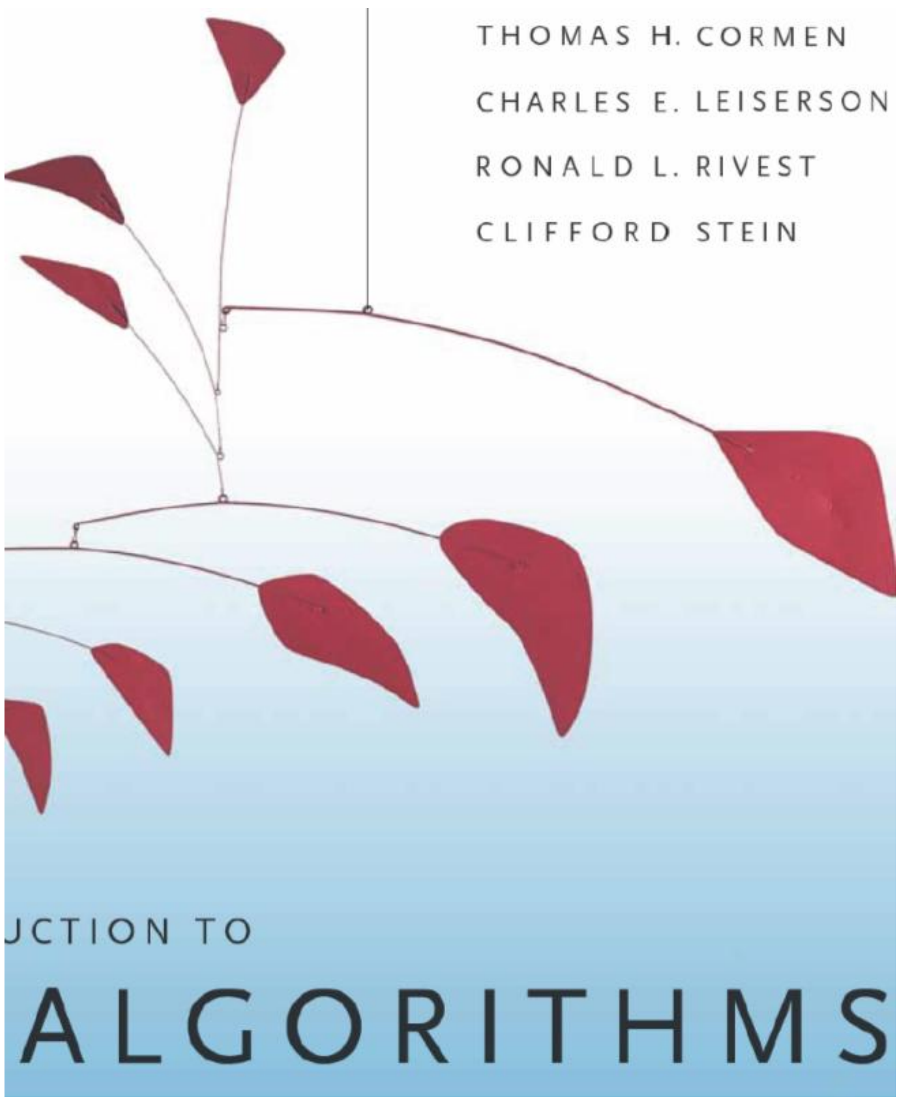


Both add(x,t) and t remain accessible.

Running time and newly allocated nodes bounded by tree height.

Balanced Trees: Red-Black Trees





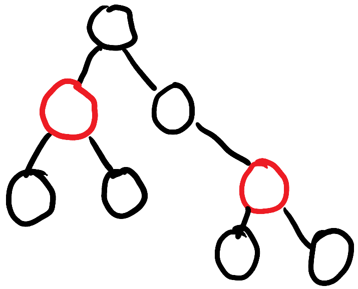
Balanced Tree: Red Black Tree

Goals:

- ensure that tree height remains at most log(size)
  add(1,add(2,add(3,...add(n,Empty())...)))   ~ linked list
- preserve efficiency of individual operations:
  rebalancing arbitrary tree: could cost O(n) work

Solution: maintain mostly balanced trees: height still O(log size)

```
sealed abstract class Color
case class Red() extends Color
case class Black() extends Color
```



```
sealed abstract class Tree
case class Empty() extends Tree
case class Node(c: Color,left: Tree,value: Int, right: Tree)
                    extends Tree
```

Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its *color*, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following ***red-black properties***:

balanced tree constraints

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

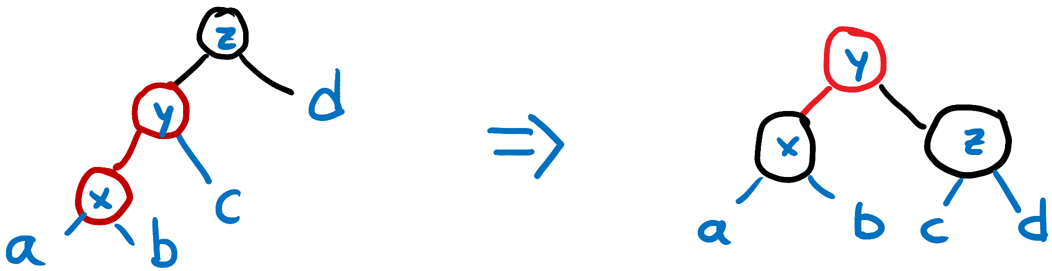
From 4. and 5.: tree height is O(size). ← O(log size)

Analysis is similar for mutable and immutable trees.

for immutable trees: see book by Chris Okasaki

Balancing

```
def balance(c: Color, a: Tree, x: Int, b: Tree): Tree = (c,a,x,b) match {
  case (Black(),Node(Red(),Node(Red(),a,xV,b),yV,c),zV,d) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
```

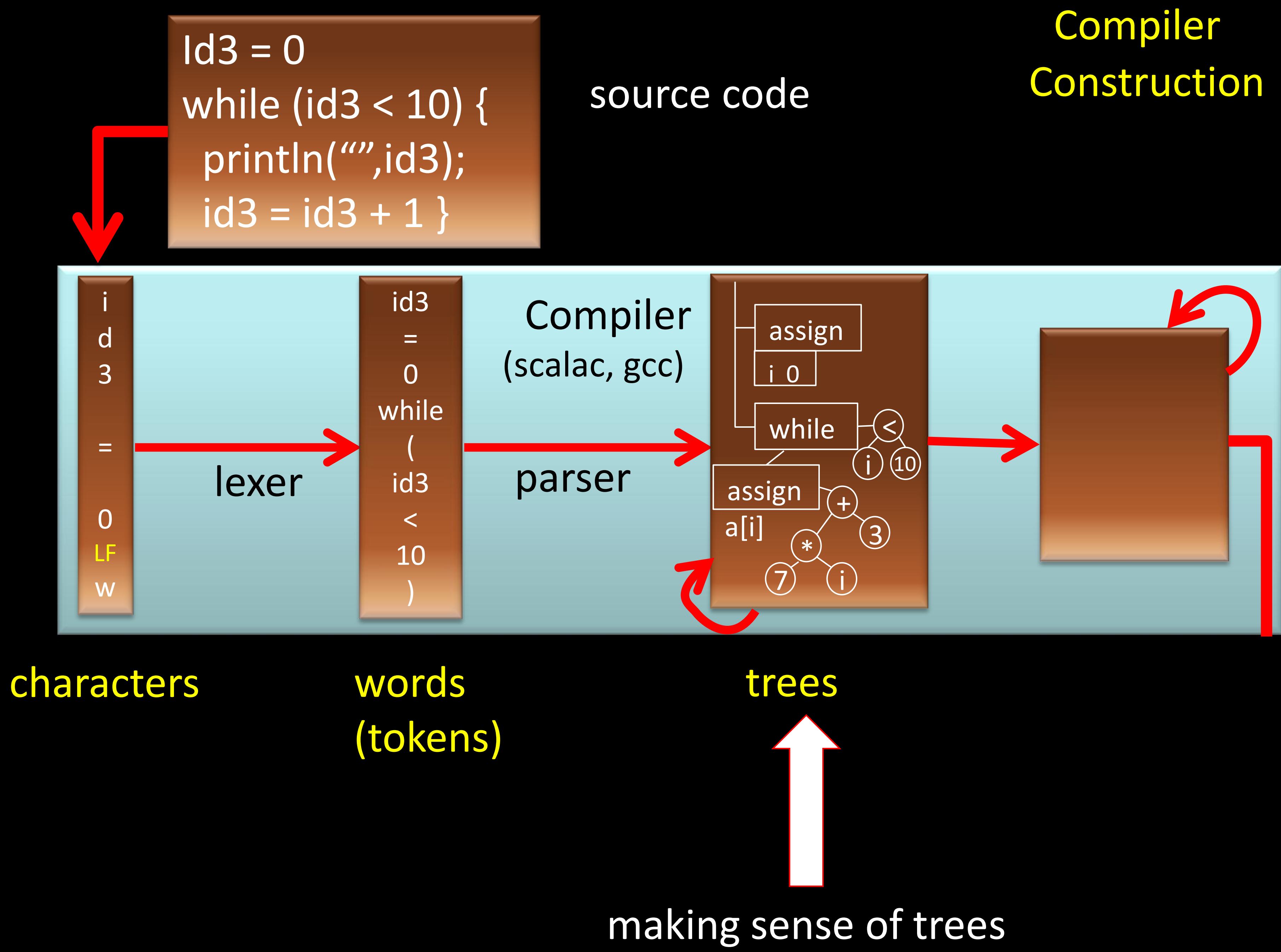



```
  case (Black(),Node(Red(),a,xV,Node(Red(),b,yV,c)),zV,d) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),Node(Red(),b,yV,c),zV,d)) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),b,yV,Node(Red(),c,zV,d))) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (c,a,xV,b) => Node(c,a,xV,b)
}
```

Insertion

```
def add(x: Int, t: Tree): Tree = {
  def ins(t: Tree): Tree = t match {
    case Empty() => Node(Red(),Empty(),x,Empty())
    case Node(c,a,y,b) =>
      if (x < y) balance(c, ins(a), y, b)
      else if (x == y) Node(c,a,y,b)
      else balance(c,a,y,ins(b))
  }
  makeBlack(ins(t))
}
def makeBlack(n: Tree): Tree = n match {
    case Node(Red(),l,v,r) => Node(Black(),l,v,r)
    case _ => n
}
```

Modern object-oriented languages (=Scala) support abstraction and functional data structures. Just use Map from Scala.


Compiler
Construction
Id3 = 0
while (id3 < 10) {
println(“”,id3);
id3 = id3 + 1 }
source code
i
d
3
=
0
LF
w
lexer
id3
=
0
while
(
id3
<
10
)
Compiler
(scalac, gcc)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
characters
words
(tokens)
trees
making sense of trees

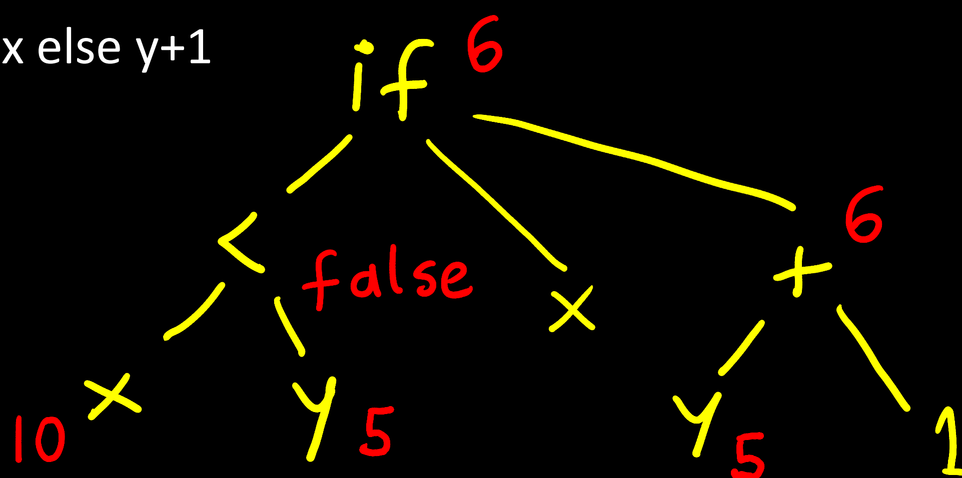
Evaluating an Expression

scala prompt:

```
>def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
>min1(10,5)
res1: Int = 6
```

How can we think about this evaluation?

x $\rightarrow$ 10
y $\rightarrow$ 5
if (x < y) x else y+1



Each Value has a Type

scala prompt:

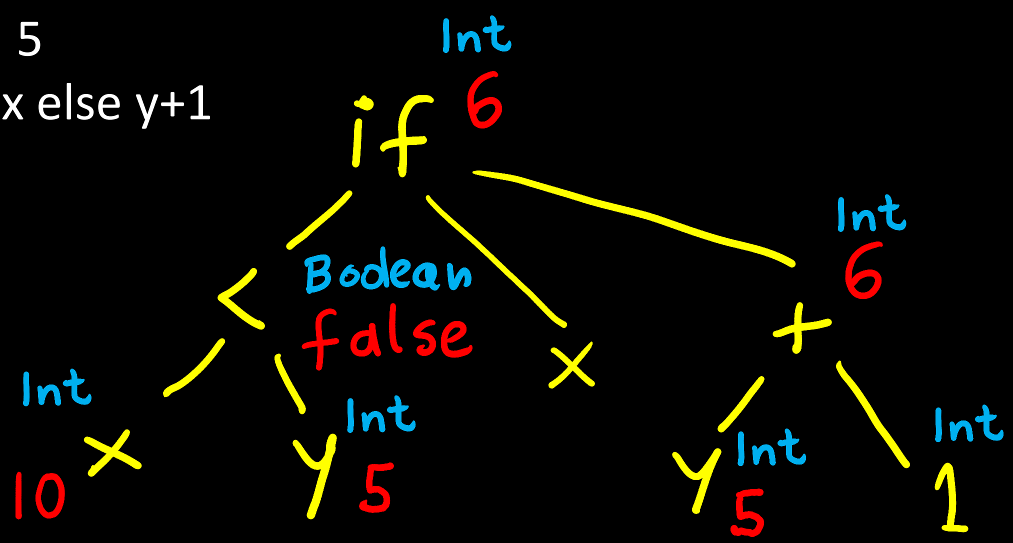
```
>def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
>min1(10,5)
res1: Int = 6
```

How can we think about this evaluation?

x : Int → 10

y : Int → 5

if (x < y) x else y+1



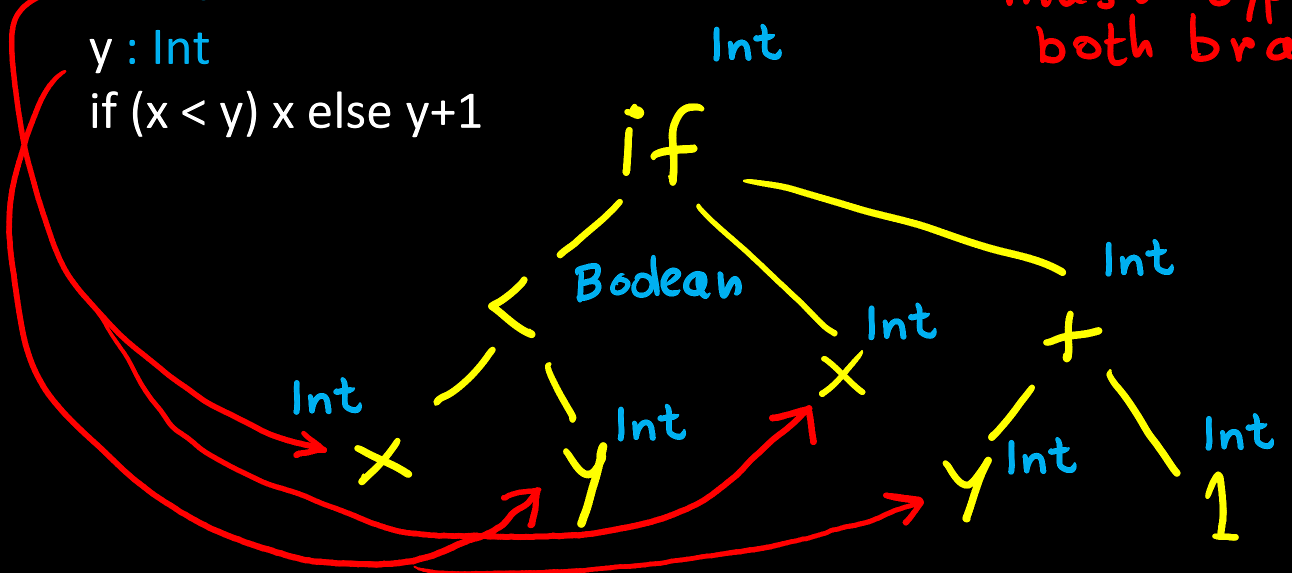
We can compute types without values

scala prompt:

```
>def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
>min1(10,5)
res1: Int = 6
```

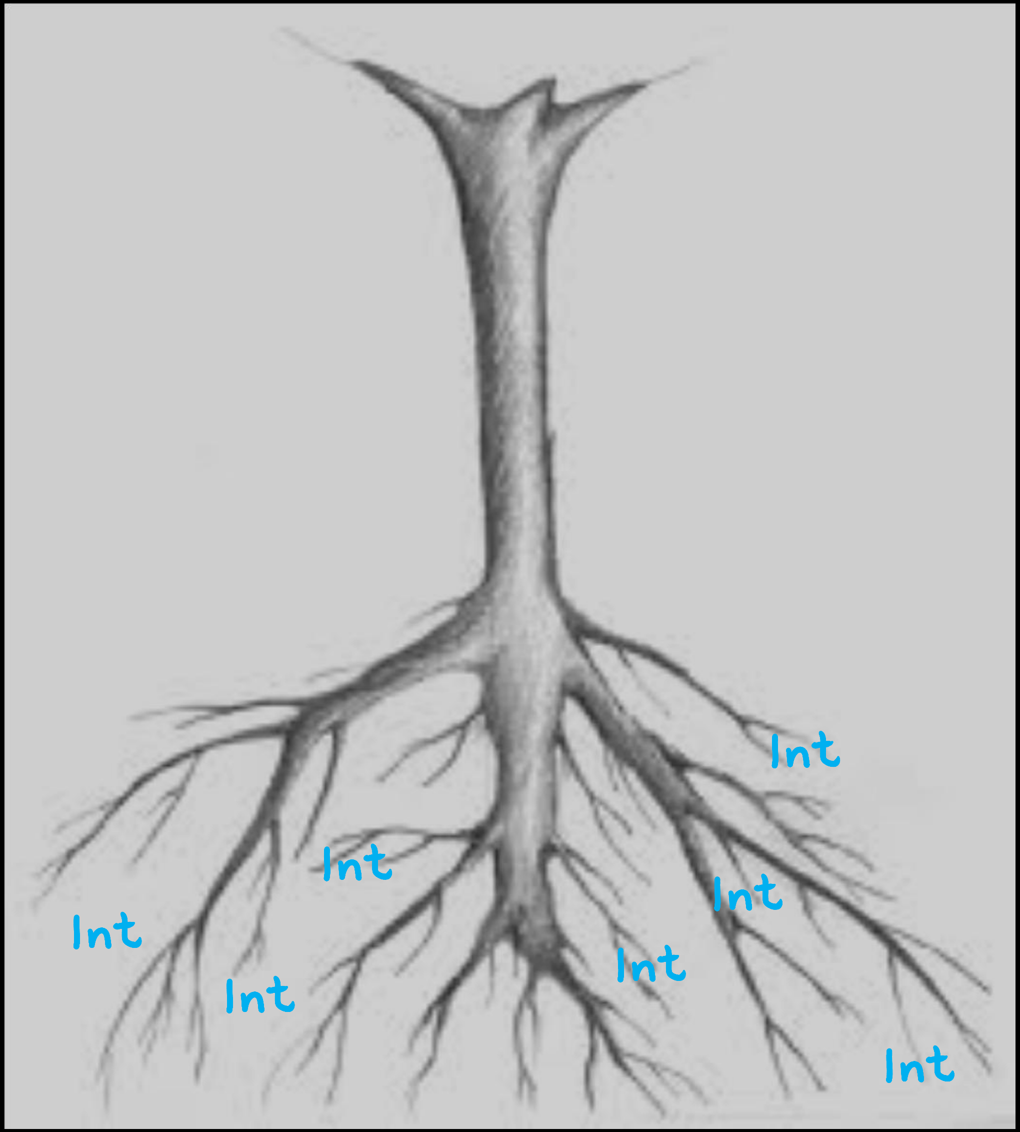
How can we think about this evaluation?

x : Int
y : Int
if (x < y) x else y+1

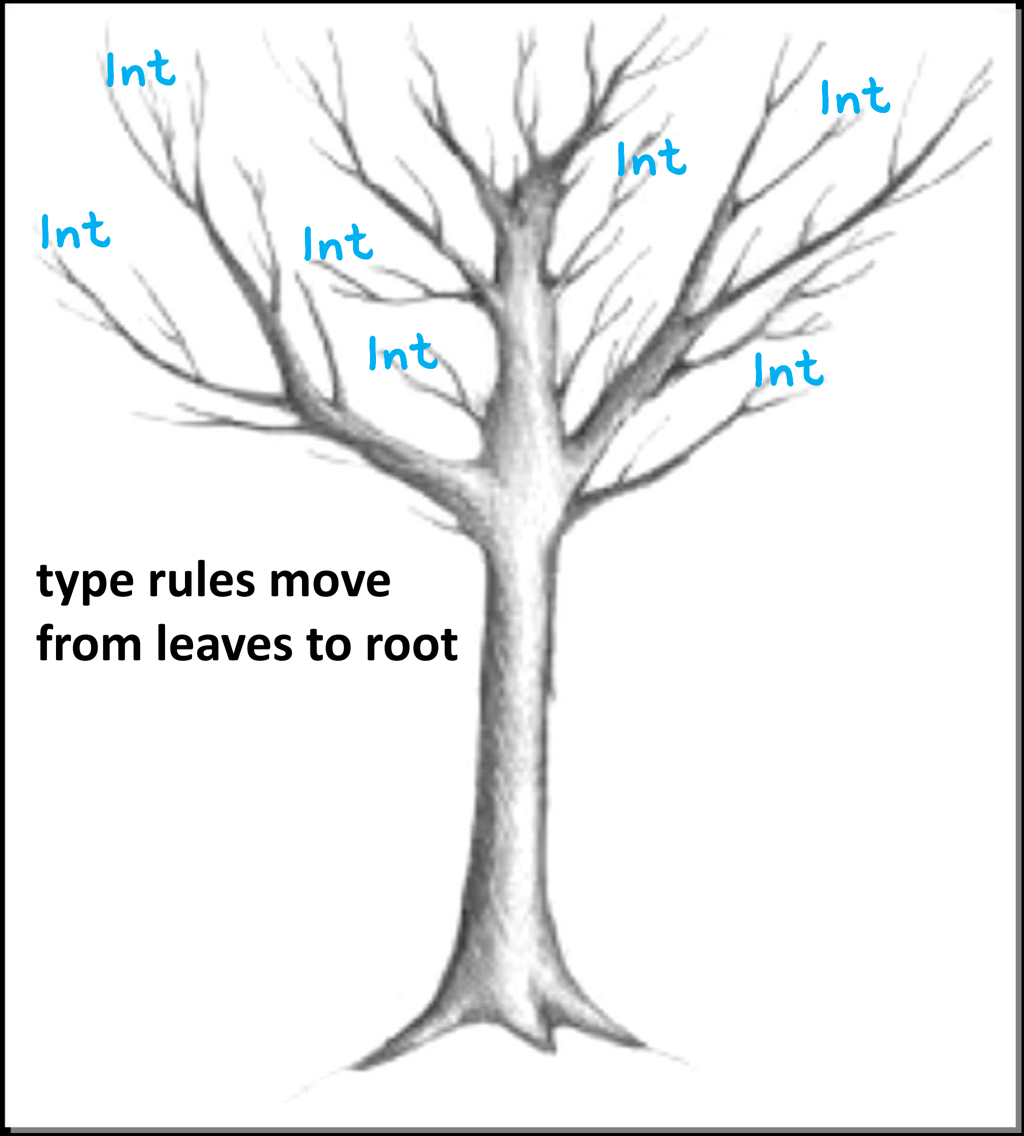


must type check both branches

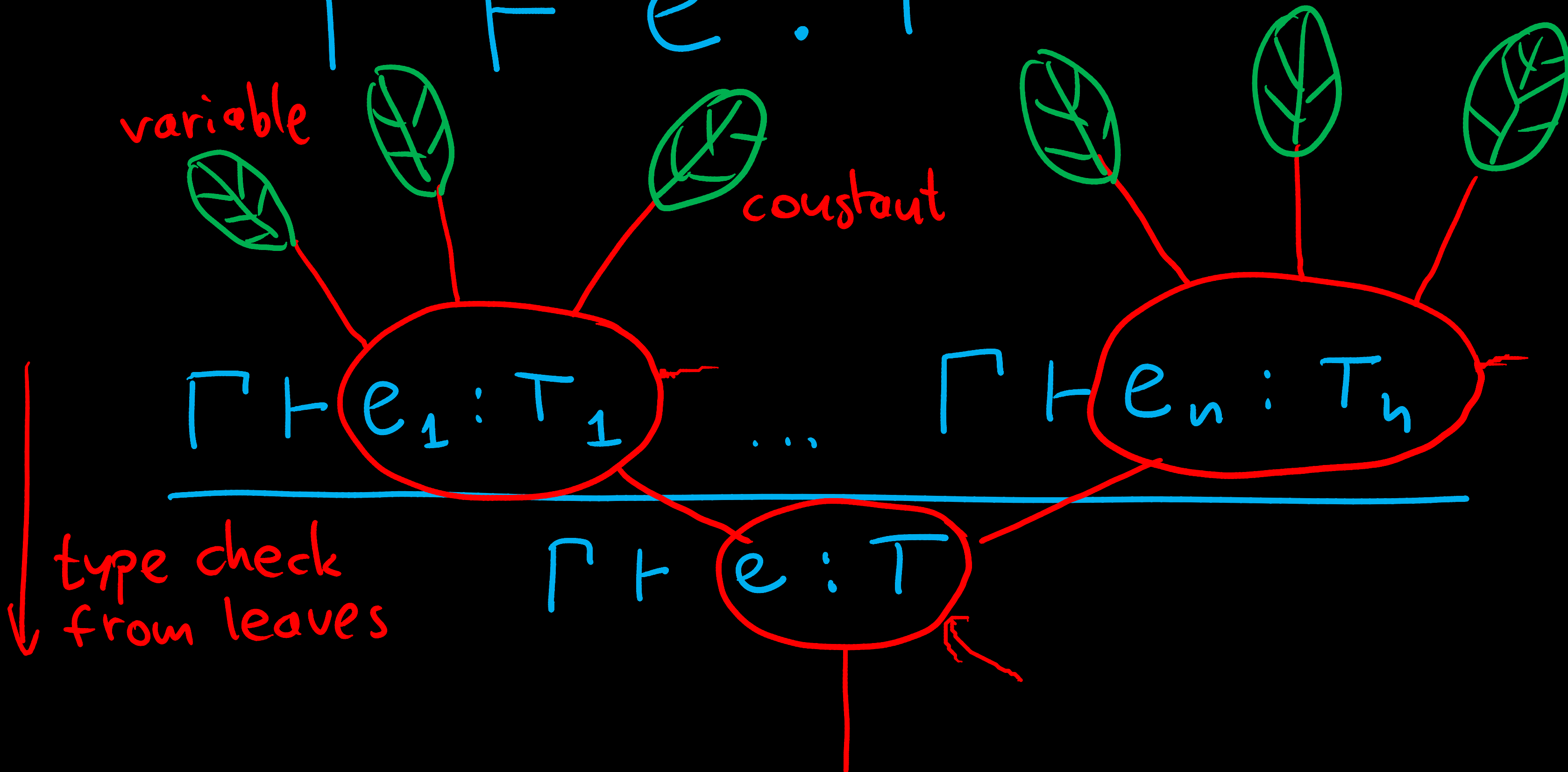
We do not like trees upside-down



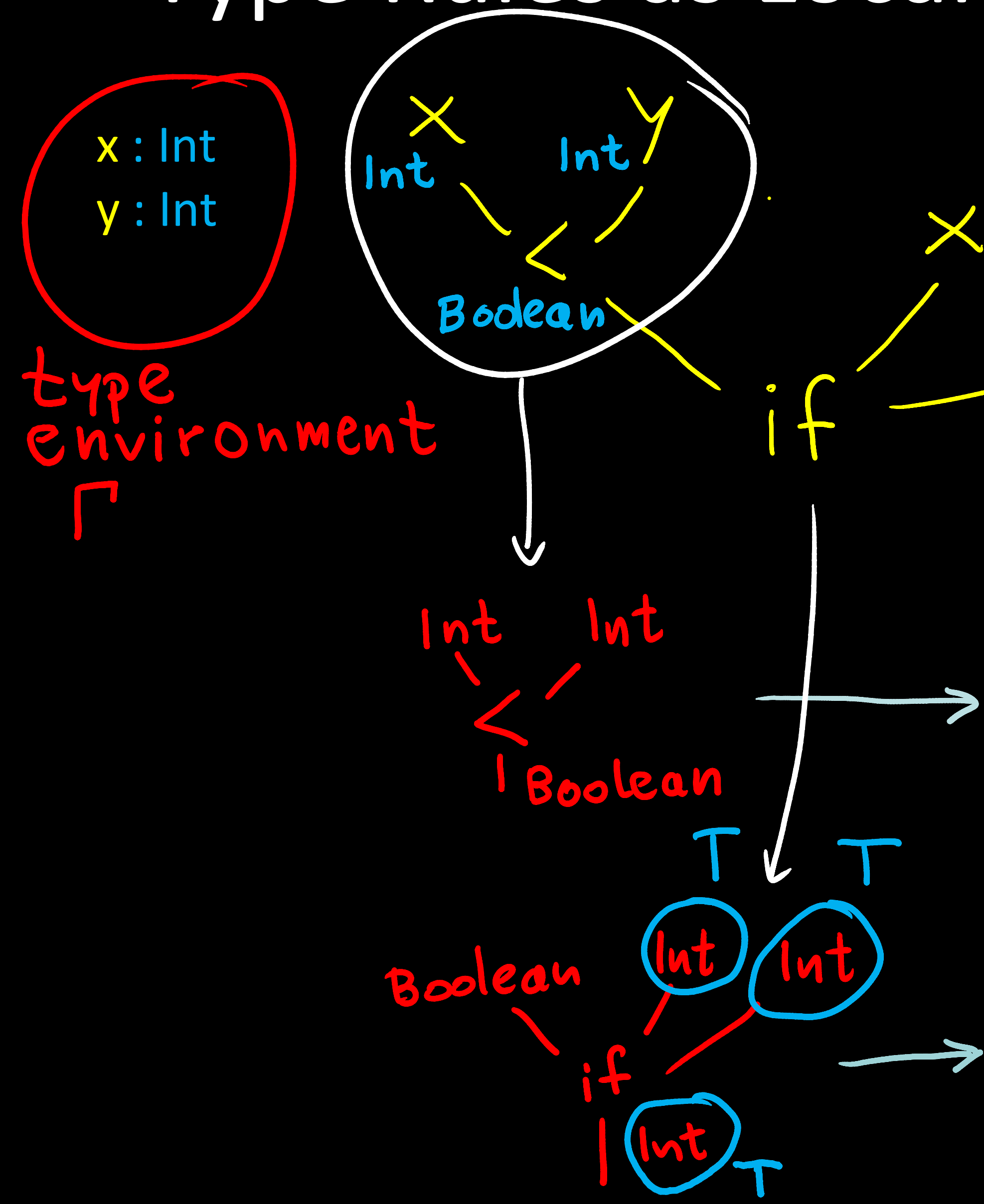
Leaves are Up



$$\Gamma \vdash e : T$$



Type Rules as Local Tree Constraints



Type Rules

$$\frac{e_1 : Int \qquad e_2 : Int}{e_1 < e_2 : Boolean}$$

for every type T, if

b has type Boolean, and ...

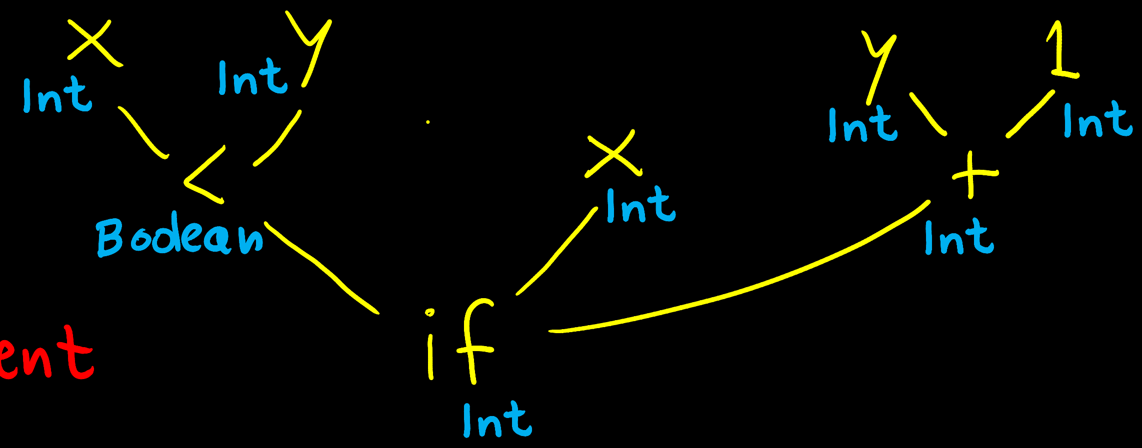
then

$$\frac{b : Boolean \qquad e_1 : T \qquad e_2 : T}{(\text{if } (b)\ e_1 \text{ else } e_2) : T}$$

Type Rules with Environment

x : Int
y : Int

type environment Γ



Type Rules

$$\frac{(x:T)\in\Gamma}{\Gamma \vdash x:T}$$

$$\frac{}{\mathrm{IntConst}(k):\mathrm{Int}}$$

$$\frac{\Gamma\vdash e_1:\mathrm{Int} \qquad \Gamma\vdash e_2:\mathrm{Int}}{\Gamma\vdash(e_1<e_2):\mathrm{Boolean}}$$

...(then) in the (same) environment Γ the expression $e_1<e_2$ has type Bool.

$$\frac{\Gamma\vdash e_1:\mathrm{Int}\qquad\Gamma\vdash e_2:\mathrm{Int}}{\Gamma\vdash(e_1+e_2):\mathrm{Int}}$$

$$\frac{\Gamma\vdash b:\mathrm{Boolean}\qquad \Gamma\vdash e_1:T\qquad\Gamma\vdash e_2:T}{\Gamma\vdash(\mathrm{if}\,(b)\ e_1\ \mathrm{else}\ e_2):T}$$

$$\Gamma \vdash e : T$$

if the free variables of e have types given by gamma,
then e (correctly) type checks and has type T

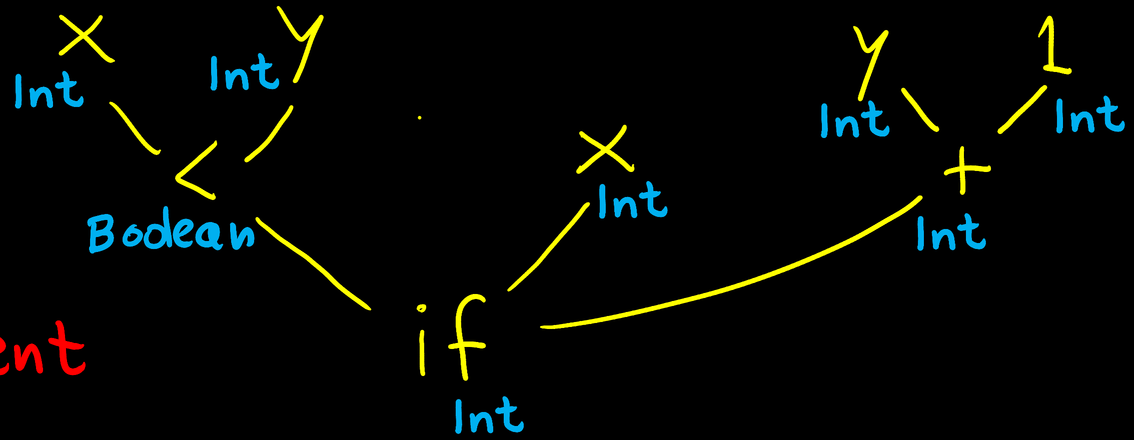
$$\frac{\Gamma \vdash e_1 : T_1 \quad \ldots \quad \Gamma \vdash e_n : T_n}{\Gamma \vdash e : T}$$

If e_1 type checks in gamma and has type T_1 and ...
and e_n type checks in gamma and has type T_n
then e type checks in gamma and has type T

Derivation Using Type Rules

x : Int
y : Int

type environment
Γ



Let $\Gamma = \{(x, Int), (y, Int)\}$

$$\dfrac{\dfrac{\dfrac{(x, Int) \in \Gamma}{\Gamma \vdash x : Int} \quad \dfrac{(y, Int) \in \Gamma}{\Gamma \vdash y : Int}}{\Gamma \vdash (x < y) : Boolean} \quad \dfrac{(x, Int) \in \Gamma}{\Gamma \vdash x : Int} \quad \dfrac{\dfrac{(y, Int) \in \Gamma}{\Gamma \vdash y : Int} \quad \dfrac{}{\Gamma \vdash 1 : Int}}{\Gamma \vdash (y+1) : Int}}{\Gamma \vdash (\text{if} (x < y)\ x \text{ else } y+1) : Int}$$

Type Rule for Function Application

We can treat operators as variables that have function type

$+ : Int \times Int \to Int$

$< : Int \times Int \to Boolean$

$\&\& : Boolean \times Boolean \to Boolean$

We can replace many previous rules with application rule:

$$\boxed{\frac{\Gamma \vdash e_1 : T_1 \quad \dots \quad \Gamma \vdash e_n : T_n \quad \Gamma \vdash f : (T_1 \times \dots \times T_n \to T)}{\Gamma \vdash f(e_1, \dots, e_n) : T}}$$

Computing the Environment of a Class

$\Gamma_0 = \{$

```
object World {
  var data : Int
  var name : String
  def m(x : Int, y : Int) : Boolean { ... }
  def n(x : Int) : Int {
    if (x > 0) p(x – 1) else 3
  }
  def p(r : Int) : Int = {
    var k = r + 2
    m(k, n(k))
  }
}
```

(data, Int),
(name, String),
(m, Int x Int → Boolean),
(n, Int → Int),
(p, Int → Int)
}

Type check each function m,n,p in this global environment

Extending the Environment

```
class World {
  var data : Int
  var name : String
  def m(x : Int, y : Int) : Boolean { ... }
  def n(x : Int) : Int {
    if (x > 0) p(x – 1) else 3
  }
  def p(r : Int) : Int = {
     var k:Int = r + 2
     m(k, n(k))
  }
}
```

$\Gamma_0 = \{$ (data, Int), (name, String), (m, Int x Int $\to$ Boolean), (n, Int $\to$ Int), (p, Int $\to$ Int) $\}$

Γ_0

$\Gamma_1 = \Gamma_0 \oplus \{(r, Int)\}$

$\Gamma_2 = \Gamma_1 \oplus \{(k, Int)\} = \Gamma_0 \cup \{(r, Int), (k, Int)\}$

Type Checking Expression in a Body

```
class World {
  var data : Int
  var name : String
  def m(x : Int, y : Int) : Boolean { ... }
  def n(x : Int) : Int {
    if (x > 0) p(x – 1) else 3
  }
  def p(r : Int) : Int = {
     var k:Int = r + 2
     m(k, n(k))
  }
}
```

$\Gamma_0 = \{$ (data, Int), (name, String), (m, Int × Int → Boolean), (n, Int → Int), (p, Int → Int) $\}$

Γ_0

$\Gamma_1 = \Gamma_0 \oplus \{(r, Int)\}$

$\Gamma_2 = \Gamma_1 \oplus \{(k, Int)\}$

Bool

$$\dfrac{\Gamma_2 \vdash k : Int \qquad \dfrac{\Gamma_2 \vdash k : Int \quad \Gamma_2 \vdash n : Int \to Int}{\Gamma_2 \vdash n(k) : Int} \qquad \Gamma_2 \vdash m : Int \times Int \to Int}{\Gamma_2 \vdash m(k, n(k)) : Int}$$

Bool ≷

Remember Function Updates

$$\{(x,T_1),(y,T_2)\} \oplus \{(x,T_3)\} = \{(x,T_3),(y,T_2)\}$$

Type Rule for Method Bodies

$$\frac{\Gamma \oplus \{(x_1,T_1),\ldots,(x_n,T_n)\} \vdash e : T}{\Gamma \vdash (\text{def } m(x_1:T_1,\ldots,x_n:T_n):T = e) : \text{OK}}$$

Type Rule for Assignments

$$\frac{(x,T) \in \Gamma \qquad \Gamma \vdash e : T}{\Gamma \vdash (x = e) : \text{void}}$$

Type Rules for Block: { var x_1:T_1 ... var x_n:T_n; s_1; ... s_m; e }

$$\frac{(\Gamma \oplus \{(x_1,T_1)\}) \oplus \ldots \oplus \{(x_n,T_n)\} \vdash \begin{array}{l} s_1 : \text{void} \\ \ldots \\ s_n : \text{void} \\ e : T \end{array} \qquad x_1,\ldots,x_n \text{ distinct}}{\Gamma \vdash \{\text{var } x_1:T_1;\ldots;\text{var } x_n:T_n; s_1;\ldots;s_n; e\} : T}$$

Blocks with Declarations in the Middle

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \{e\} : T}$$

just expression

$$\frac{}{\Gamma \vdash \{\} : \mathrm{void}}$$

empty

$$\frac{\Gamma \oplus \{(x, T_1)\} \vdash \{t_2; \ldots; t_n\} : T}{\Gamma \vdash \{\mathrm{var}\ x : T_1; t_2; \ldots; t_n\} : T}$$

declaration is first

$$\frac{\Gamma \vdash s_1 : \mathrm{void} \qquad \Gamma \vdash \{t_2; \ldots; t_n\} : T}{\Gamma \vdash \{s_1; t_2; \ldots; t_n\} : T}$$

statement is first

Rule for While Statement

$$\frac{\Gamma \vdash b : \text{Boolean} \qquad \Gamma \vdash s : \text{void}}{\Gamma \vdash (\text{while}\ (b)\ s) : \text{void}}$$

Rule for Method Call

```
class T0 {
  ...
  def m(x1:T1,...,xn:Tn):T = {
    ...
  }
  ...
}
```

$$\frac{\Gamma \vdash x : T_0 \qquad \Gamma \vdash T_0.m : T_0 \times T_1 \times \ldots \times T_n \to T \qquad \overset{\forall i \in \{1,2,\ldots,n\}}{\Gamma \vdash e_i : T_i}}{\Gamma \vdash x.m(e_1,\ldots,e_n) : T}$$

Example to Type Check

object

~~class~~ World {
  var z : Boolean
  var u : Int
Γ_0 def f(y : Boolean) : Int {
    z = y $\Gamma_1 = \Gamma_0 \oplus \{(y, \text{Boolean})\}$
    if (u > 0) {
      u = u – 1
      var z : Int
      z = f(!y) + 3
      z+z
    } else { 0 }
  }
}

$\Gamma_0 = \{$
$(z, \text{Boolean}),$
$(u, \text{Int}),$
$(f, \text{Boolean} \to \text{Int})\}$

$$\frac{\Gamma_1 \vdash z : \text{Boolean} \quad \Gamma_1 \vdash y : \text{Boolean}}{\Gamma_1 \vdash (z = y) : \text{void}}$$

$$\frac{\quad}{\Gamma_1 \vdash}$$