NTNU – Trondheim
Norwegian University of
Science and Technology

Type judgments

Where we are, conceptually

- Last time, we went through a way to see program execution as proof construction in a restricted logic
  - We're primarily stealing some notation from that exercise
  - Specifically, we'll portray type judgments as a similar sort of inference
- Before that, we went through the connection between traversing a syntax tree and inherited/synthesized attributes of its internal nodes

Where we are, textually

- Bouncing back and forth between ch. 5 / 6, I’m afraid
- There are bits about types in both of them
- There are bits in both of them which aren’t about types
    - As stated at the very beginning, I’m trying to complement the book with intuitions
    - pro: it provides several different ways to look at the subject
    - con: it doesn’t come out in the same order as the table of contents
- The stuff we’re presently covering is the foggiest part
- I’ll aim to squeeze in a summary to connect the dots as soon as we get through 6

  (For the meantime, this week draws on 5.3, 6.3 and 6.5)

A declaration

(This is a walkthrough of Fig.5.17 in the Dragon)

T → B C

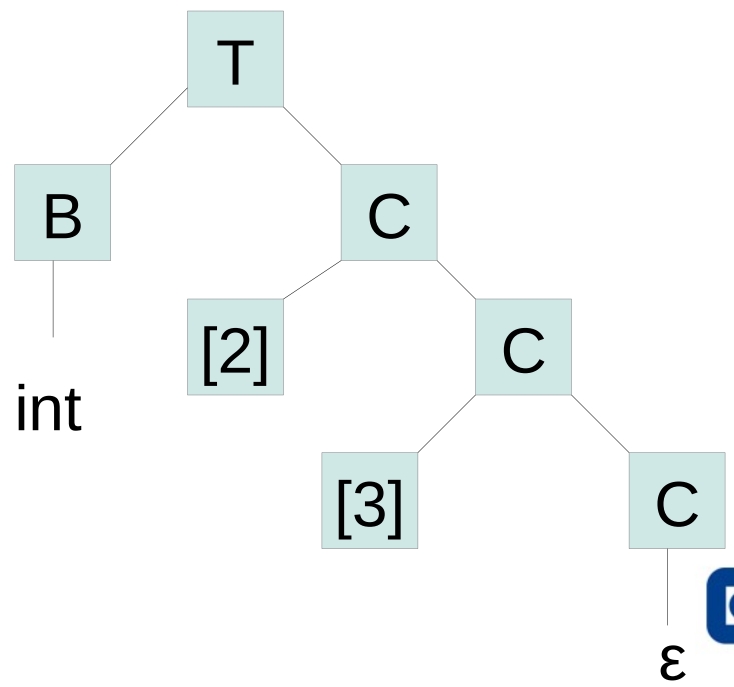
B → int | float

C → [num] C | ε

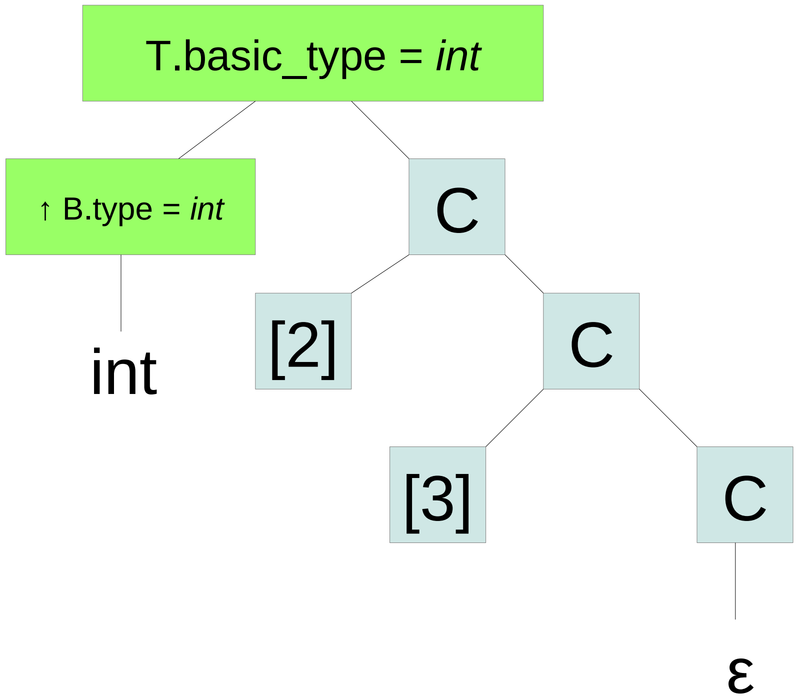
permits

int[2][3]

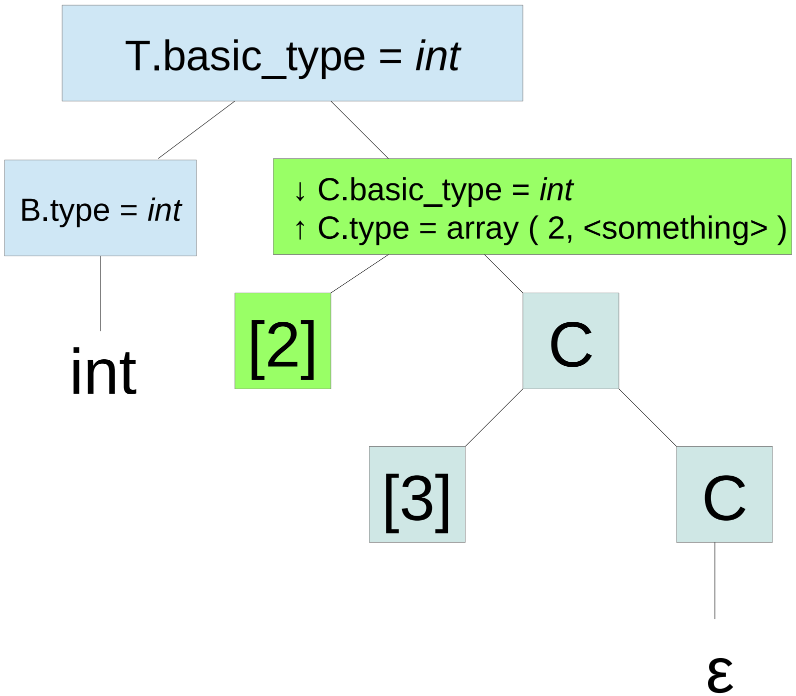
to generate



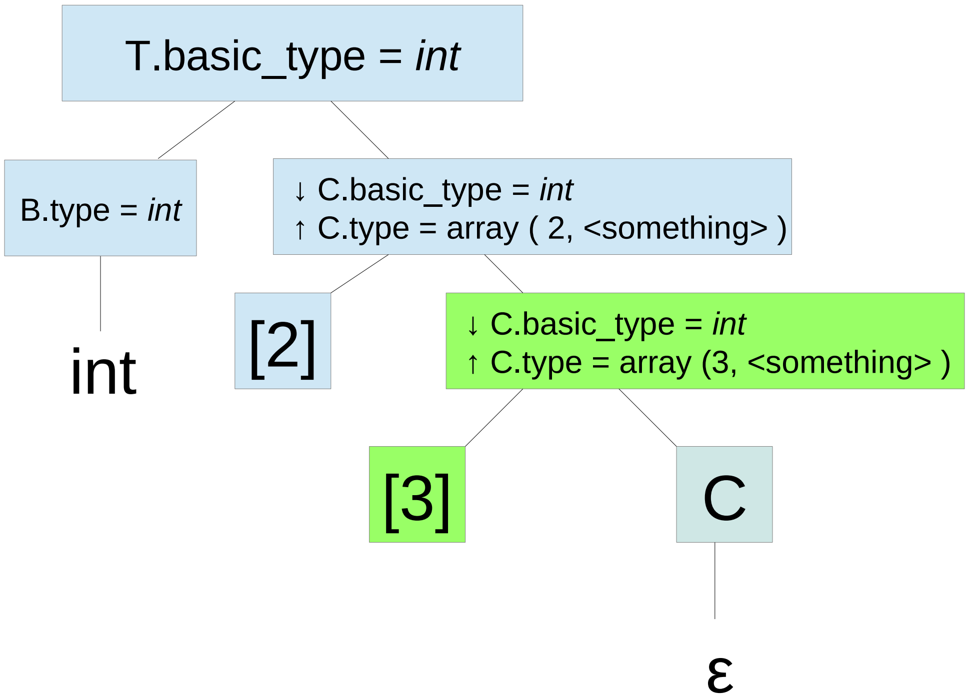
L-attribution, step 1



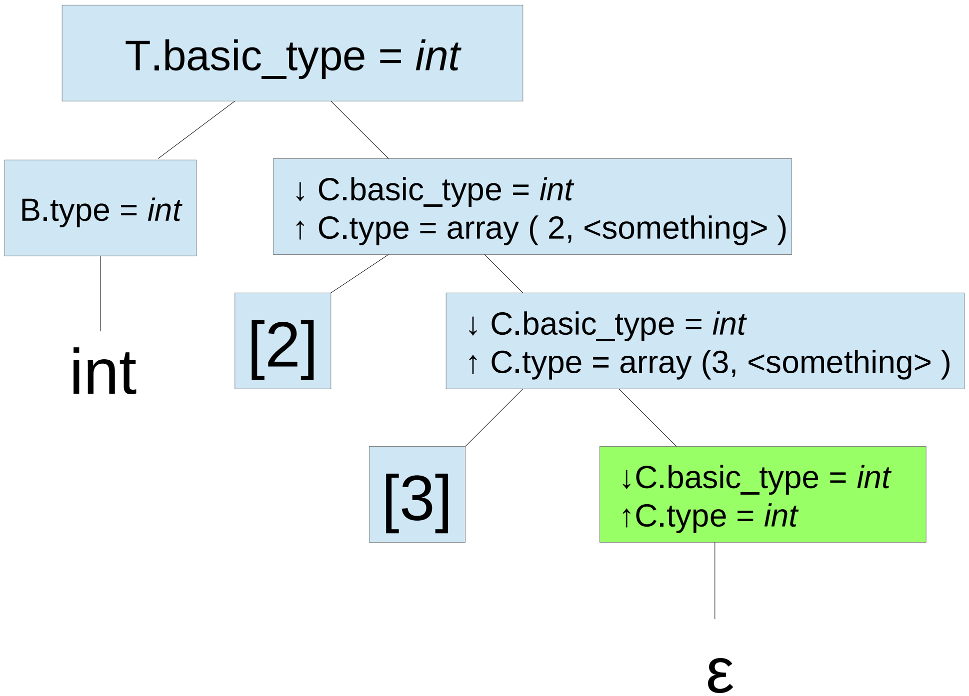
L-attribution, step 2



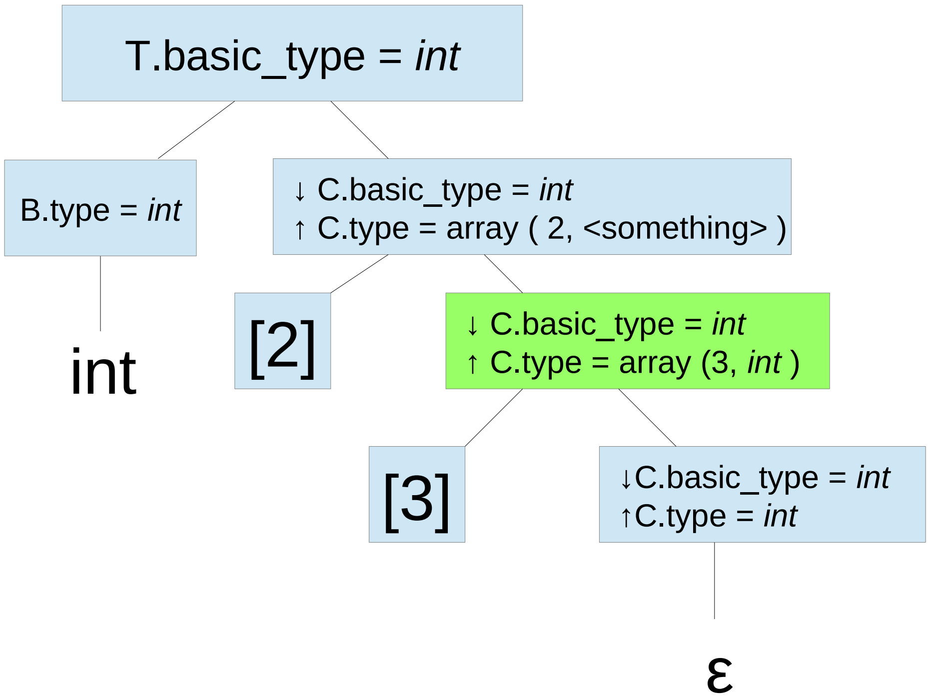
L-attribution, step 3



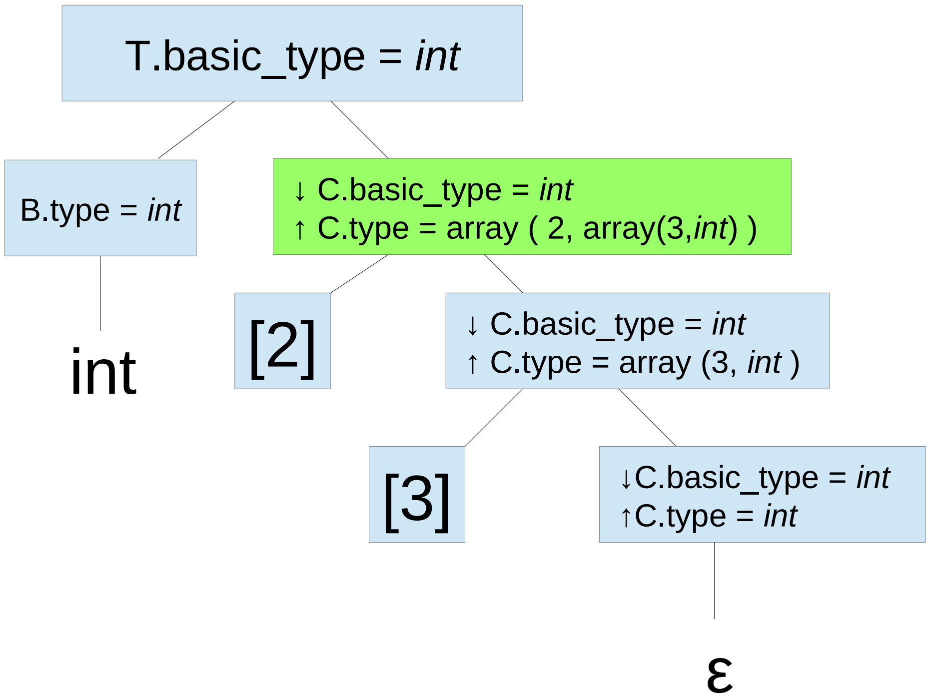
L-attribution, step 4



L-attribution, step 5

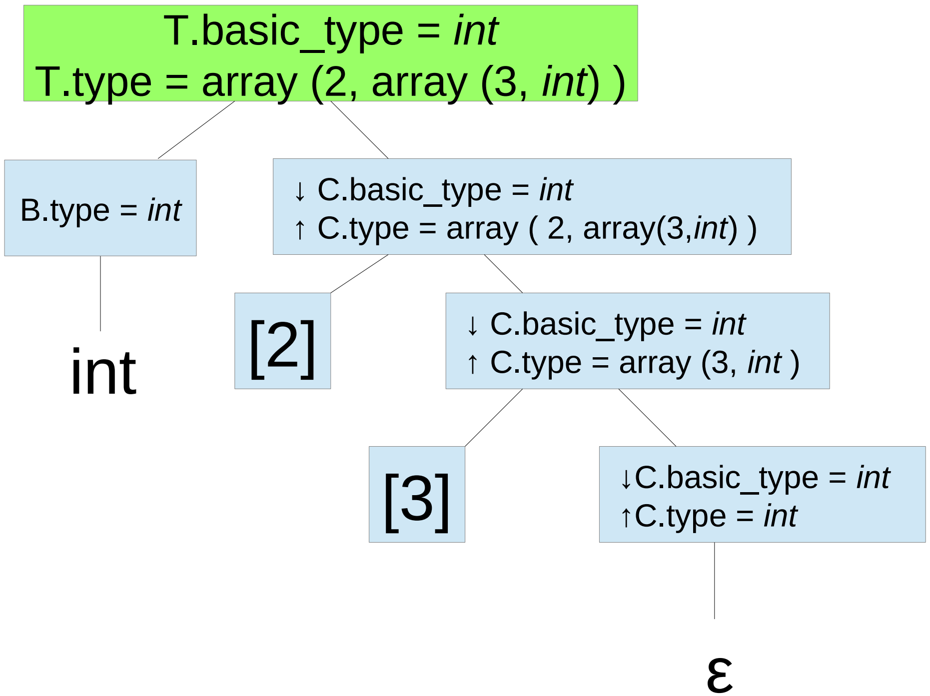


L-attribution, step 5



NTNU – Trondheim
Norwegian University of Science and Technology

L-attribution, step 6



Attribution rules

T → B C	Synthesize T.basic_type Let C inherit T.basic_type Synthesize T.type = C.type
B → int	B.type = *int*
B → float	B.type = *float*
C_0 → [num] C_1	Let C_1 inherit C_0.basic_type Synthesize C_0.type = array (num, C_1.type)
C → ε	Synthesize C.type = C.basic_type

NTNU – Trondheim
Norwegian University of Science and Technology

A smaller example

- Take these ternary expressions:

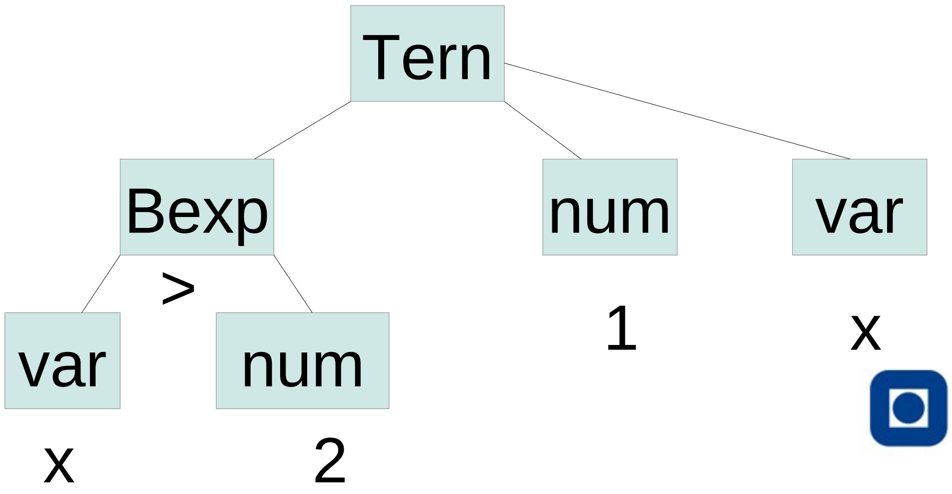
  Tern → Bexp ? Exp : Exp

  Bexp → true | false | Exp > Exp

  Exp → num | var

  and create the parse tree for

  x>2 ? 1 : x

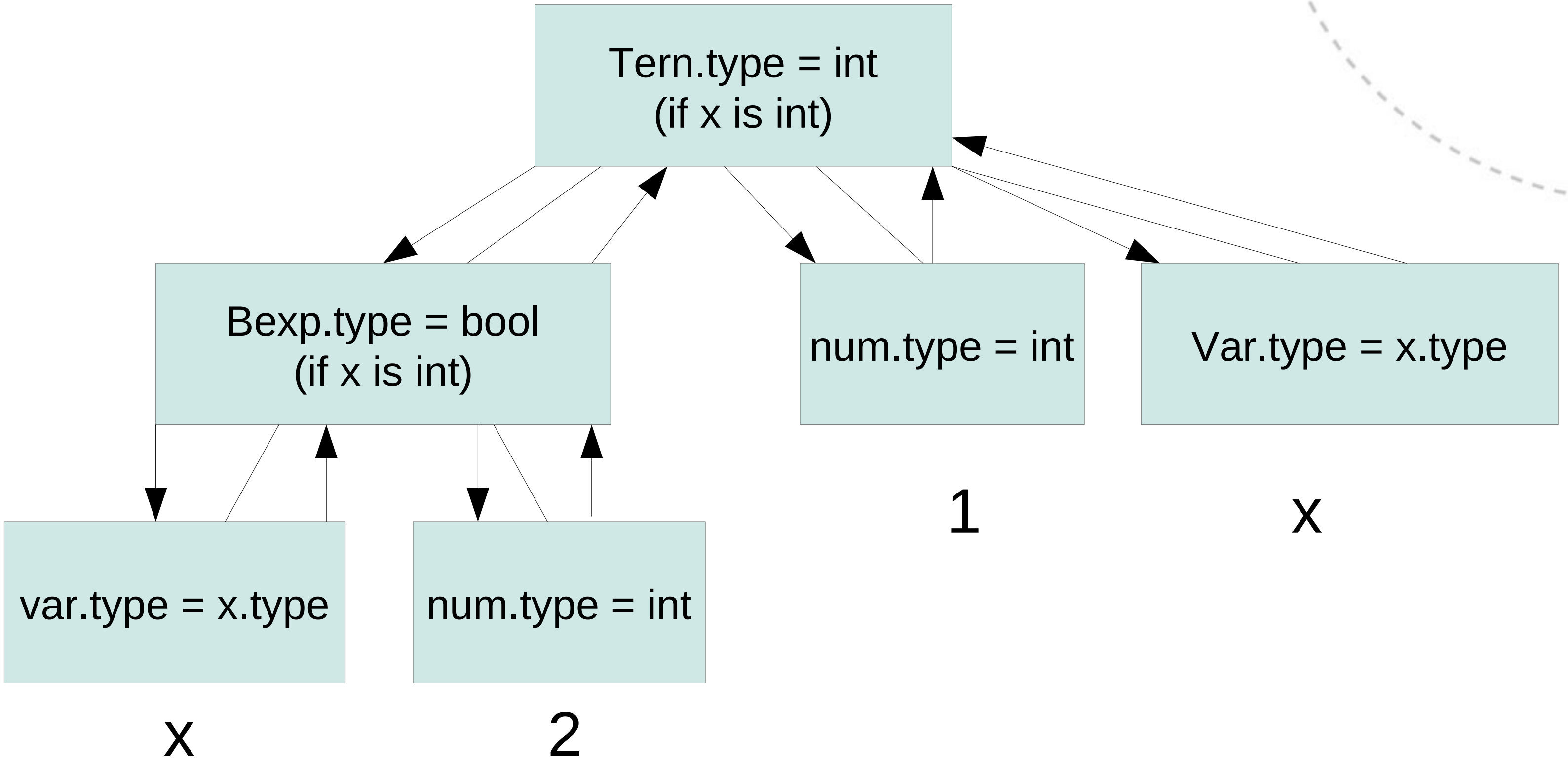


A smaller example

- To verify that it's a valid expression,

Tern → Bexp ? Exp1 ; Exp2	visit Bexp, synthesize bool synthesize Exp1.type synthesize Exp2.type enforce Exp1.type = Exp2.type
Bexp → true | false	synthesize bool
Bexp → Exp1 > Exp2	synthesize Exp1.type synthesize Exp2.type enforce Exp1.type = Exp2.type
Exp → num	Exp.type = num.type
Exp → var	Exp.type = var.type

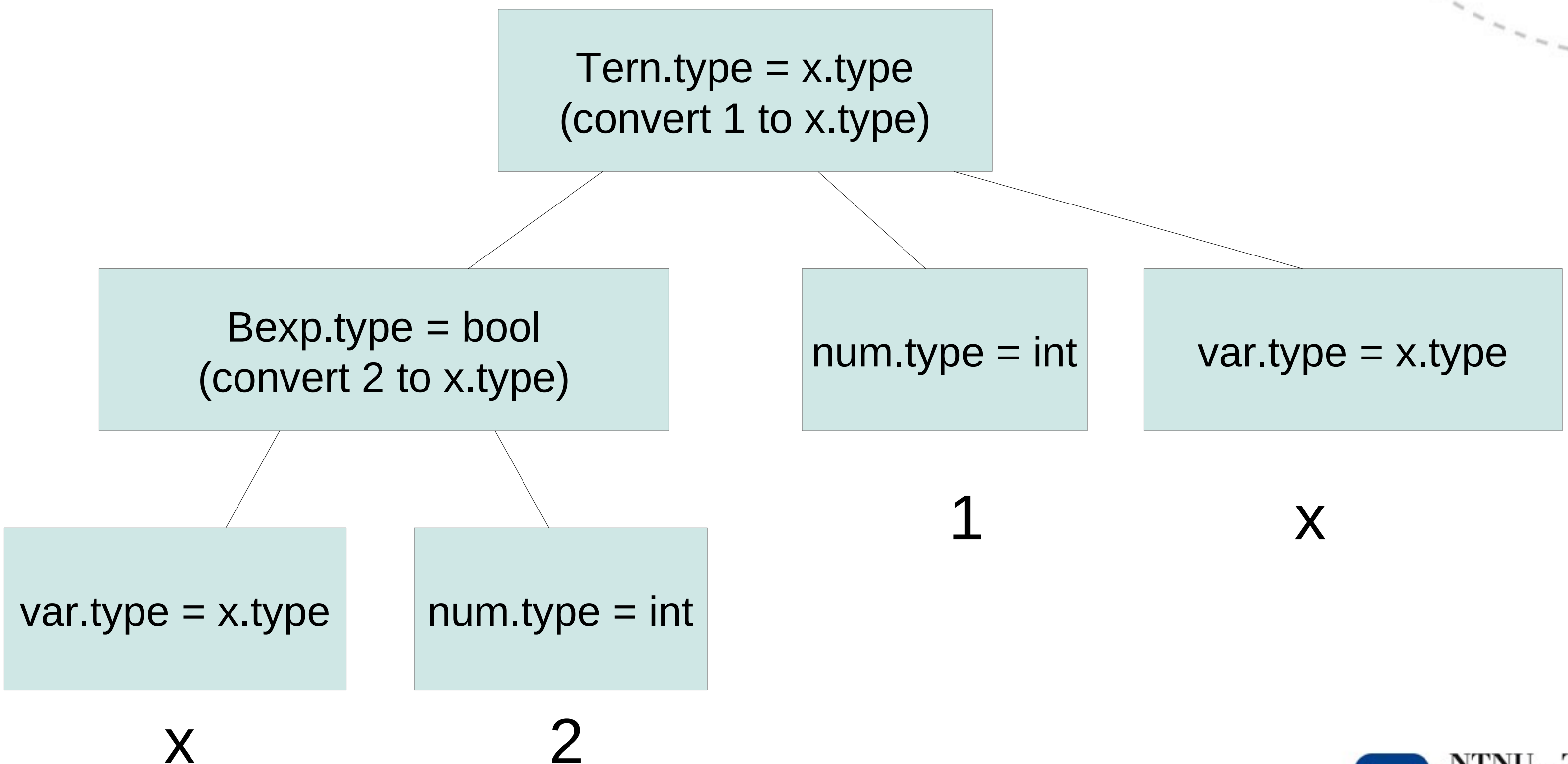
Very Strictly, in traversal order



(Strictly because we require x to be an int)

More relaxed

Say we allow conversion from int to x.type (whatever it is):



Disregarding the order

- For the strict interpretation, we could write

$$\frac{\text{Bexp : bool} \qquad \text{Exp1 : T} \qquad \text{Exp2 : T}}{\text{Bexp ? Exp1 ; Exp2 : T}}$$

and

$$\frac{\text{Exp1 : T} \qquad\qquad \text{Exp2 : T}}{\text{Bexp : bool |- Exp1 > Exp2 : bool}}$$

to capture the ideas that

  - Bexp is boolean when Exp1 and Exp2 have the same type T
  - Bexp ? E1 ; E2 has type T when E1 and E2 have the same type T

Proof tree

x : T2   2:T2
(x > 2) : bool   1:T1   x:T1
            (x > 2 ? 1 ; x) : T1

and get a substitution consistent with the rules if T1=T2=int:

x : int   2: int
(x > 2) : bool   1 : int   x : int
            (x > 2 ? 1 ; x) : int

(The presence of x in both sub-expressions forces 1,2, to have the same type)

Another proof tree

Changing the expression a little

y : T2 3.14:T2
____
(y > 3.14) : bool 1:T1 x:T1
____
(y > 3.14 ? 1 ; x) : T1

a consistent substitution might be T1=int, T2=float:

y : float 3.14: float
____
(y > 3.14) : bool 1 : int x : int
____
(y > 3.14 ? 1 ; x) : int

(T1 and T2 aren't necessarily the same)

In general

- We can attach static type semantics to syntax in the format

  $$\frac{H1 \vdash S1 : T1 \quad \ldots \quad Hn \vdash Sn : Tn}{H0 \vdash S0 : T0}$$

  and let
  - Hx be conjectures to prove,
  - Sx be parts of syntax expressions
  - Tx be the inferences of type information

Attribute grammars vs. static natural semantics

- In terms of traversal ordering, this corresponds to inputs (derived from the statement), and outputs (from the inference process)

$$\frac{H1 \mid\!- S1 : T1 \quad \ldots \quad Hn \mid\!- Sn : Tn}{H0 \mid\!- S0 : T0}$$

i.e., start from a conjecture, work through all its premises, conclude with the derived information

What are the H-s?

- Hypotheses. We could write out the reasoning in full,

$$\dfrac{\dfrac{y : T2 \qquad 3.14 : float}{y : float \vdash (y > 3.14) : bool} \qquad \vdash 1 : int \qquad x : int \vdash x : T1}{y{:}float, x{:}int \vdash (y > 3.14\ ?\ 1\ ;\ x) : T1}$$

to verify that what we hypothesized ("y is float, x is int") is consistent with the schema in at least one substitution of T1, T2

Why I prefer this notation

- It doesn't mix implementation (traversal order) with definition (rules of the type system)
- The attribute grammar approach is a special case of inference rules anyway

They’re the same when...

1) There are no missing definitions
   Everything in the outputs is also found from an input somewhere
2) There are no missing rules
   Each syntax construct must have an applicable rule
3) It’s deterministic
   There is only one applicable rule for each syntax construct
4) There are no constraints
   Inputs are just variables
5) There are no links
   No variables appear in several input positions
6) There is nothing dynamic
   Constructs in premises are strictly parts of the construct in the conclusion

Don't memorize that list (unless you want to)

- We will only look at cases where these inference rules could be exchanged for a tree traversal plan
- I just want to introduce the notation
  - It is used elsewhere in the literature
  - It can describe type information without pulling the details of attribution order into the picture all the time
- It would be downright cruel to set up problems that cannot be equally well expressed the way our book does it.

So, what’s a type judgment?

- It’s a claim about a statement, written

  |- E : T

  which reads “E is a well-typed construct of type T”
- *Type-checking a program* P requires demonstrating that |- P : T for a type T
- It can be done by traversal and attribution
- It can be done by some other logical inference engine

Honestly

- We won't be *implementing* type checking, our toy language has almost nothing in the way of types
- As far as this class goes, we'll do as we do with the bottom-up parsing schemes, as long as you can
  - Read and understand inference rules
  - See that they can be implemented by tree traversal and attribution

  there is no need to split hairs over the β-s and γ-s

- The valuable takeaway is to build a vocabulary that lets you make an informed guess about how types might be handled by your favorite programming language

Next up

- Next time, we'll
  - chuck together a bunch of inference rules for various basic things that are common in many languages

  and talk a bit about
  - static vs. dynamic types
  - the *strength* of a type system
  - what it means that one thing is equal to another