Type Systems

Lots of ways to go wrong…

And finally, I cannot tell you all the things whereby ye may commit programming errors; for there are divers ways and means, even so many that I cannot number them.

(paraphrasing with apologies to King Benjamin)

Lots of ways to go wrong…

- Things we already catch in our interpreter:
  - Syntax errors - caught by the parser
  - Semantic errors - unbound variables
  - Type errors - checked at runtime (addition must be between two numbers, only functions can be called, etc.)

- Type systems are a way of catching certain kinds of errors

What is a type?

What is a type?

- A type is
  - A set of values
    (each with their own representation)
  - A set of legal operations on those values

Enforcing correct use of types helps prevent misinterpreting bits!

Interpreting bits

01000010010110010101010100110001

What do these 32 bits mean?

Interpreting bits

- Storing values = encode and write
- Using values = read and interpret

} Had better be the same!

01000010010110010101010100110001

How do we check types?

Dynamic Type Checking

- At run time
- Store additional type information with the actual data
- Use the type information to know how to interpret the bits
- Errors are reported at the time the illegal operation occurs

Static Type Checking

- At compile time (before running)
- Figure out the type of everything *before every running the program*
- Make sure types are used correctly
- Errors are reported before program begins running

Advantages of static type checking

- Catch some errors at compile time, not run time
- Programmer-written test suites may be incomplete,
  but type errors can still at least be caught
- Documentation
- Compilers / interpreters can use the information for efficiency

Can we always do it?

What type does this return?

```
(if (> x 0)
    42
    “hello”)
```

Fundamental tradeoff

- Over-approximation” = reject programs that don’t have type errors
- Under-approximation = allow through programs that have type errors
- Must balance over-approximation / under-approximation and available information

Ways to get more information

- Have the programmer tell you (e.g., manifest typing)
- Cripple the language - remove things that you can't check statically
- Spend more computation to determine (but still don't be a burden)

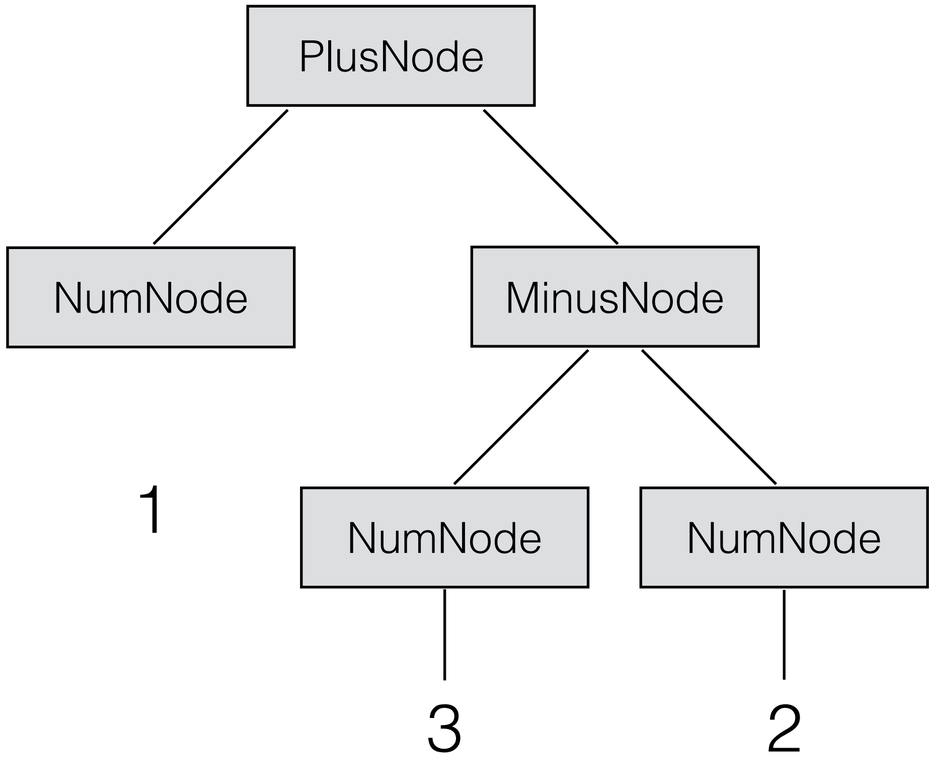
And in the end there still may be things you can't check
(some languages inject some dynamic checking)

Type systems

- A “type system” consists of
  - A set of types
  - A set of rules for figuring out and enforcing types (“type judgments”)
  - An algorithm for applying the rules

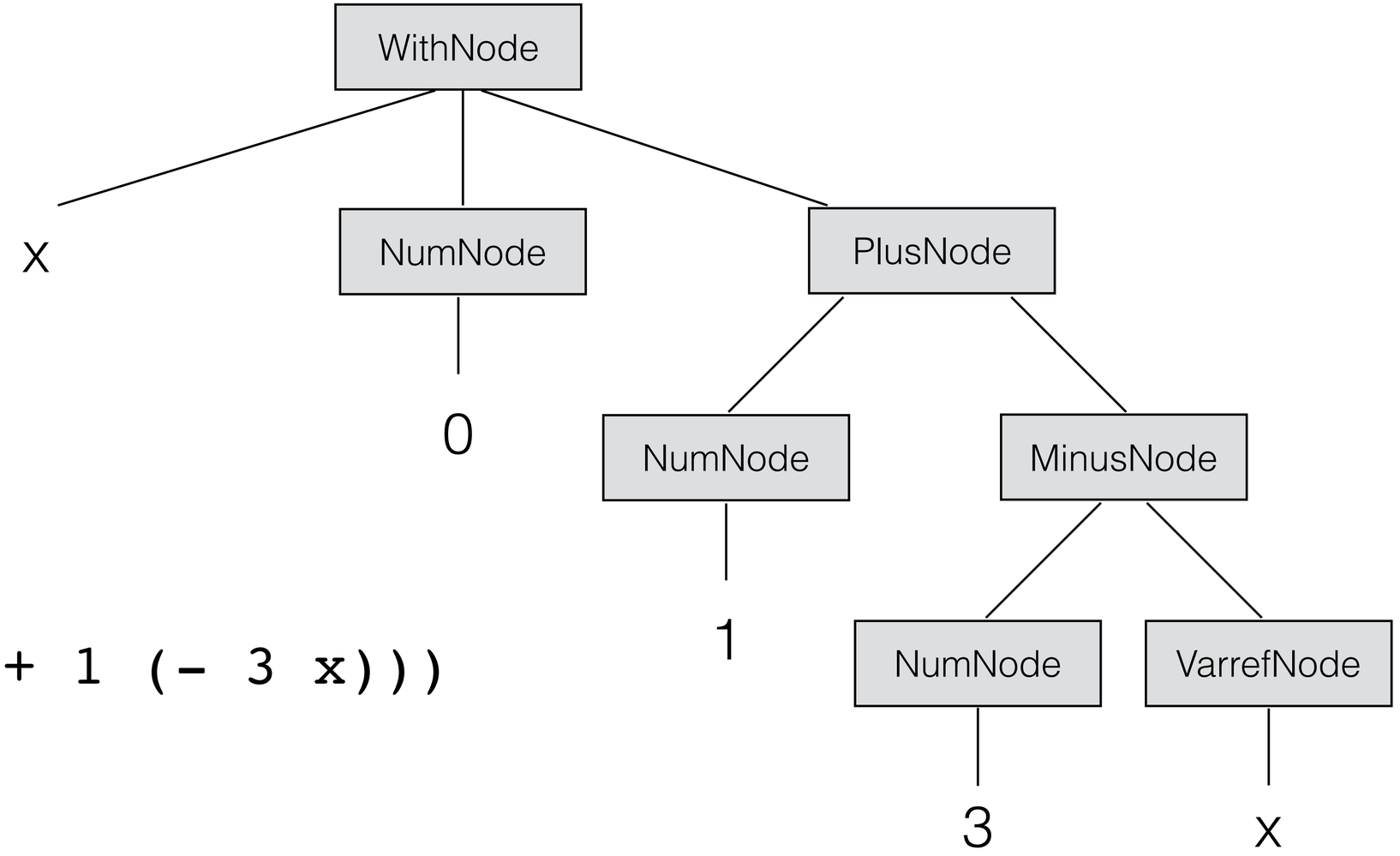
A simple type checker

- Recursively traverse the AST, and at each level:
  - Recursively type check and determine the types of any subexpressions
  - Verify that types are being used correctly in this expression
  - Return the type of the result of this expression




```
(+ 1 (- 3 x))
```

Another example



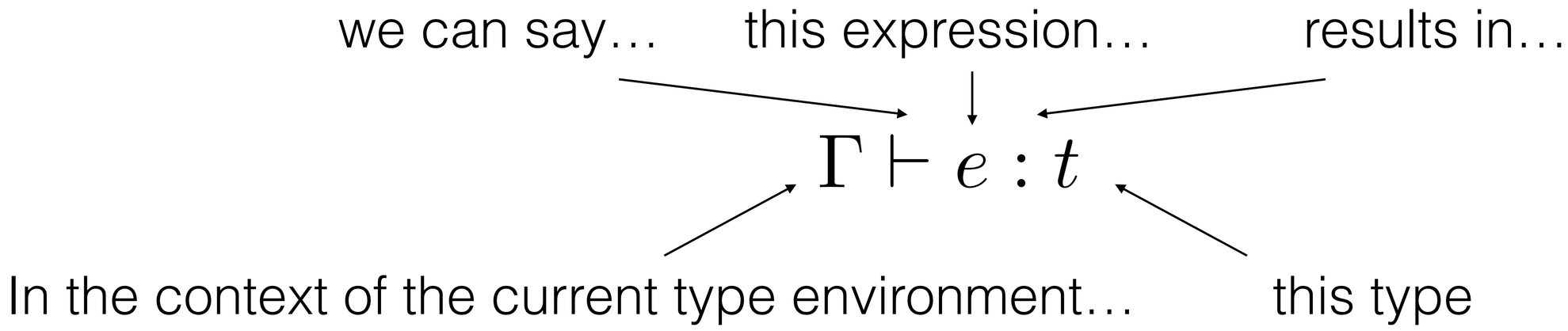

```
(with x 0 (+ 1 (- 3 x)))
```

Type environments

- To store the types of variables, use a **type environment**
  (binds identifiers to types, not values)
- Recursively pass type environments during type checking the same way we passed value environments during evaluation
- When the variable is created, store its type
  (declared or inferred from the initialization)
- When the variable is referenced, look it up in the type environment to get its type (not its value)

Type judgements

- Think about it as “proving” the types of things
- If … we can say that the type of … is …
- Notation:



Chaining type judgements

- Sometimes we can only determine the type of an expression if other things hold
- Notation:

$$\frac{\text{If these things hold…}}{\text{We can say this}}$$

Examples
(whiteboard)