```
e0, e1, e2 :: Expr
e0 = Add (Num 4.0)  (Num 2.9)
e1 = Sub (Num 3.78) (Num 5.92)
e2 = Mul e0 e1
```

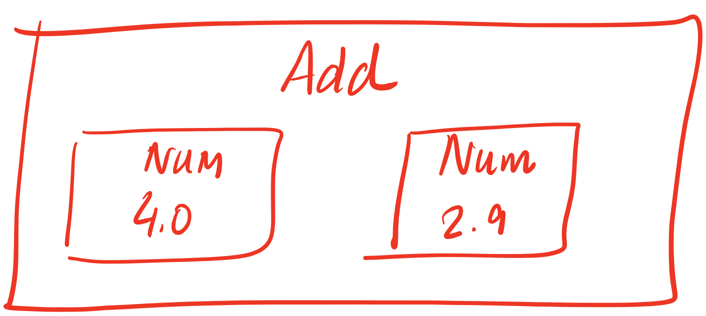
EXERCISE: Expression Evaluator

Write a function to *evaluate* an expression.

```
-- >>> eval (Add (Num 4.0)  (Num 2.9))
-- 6.9
```

```
eval :: Expr -> Float
eval e = ???
```

data Expr = Num Double
| Add Expr Expr



MIDTERM ON MONDAY 2/8

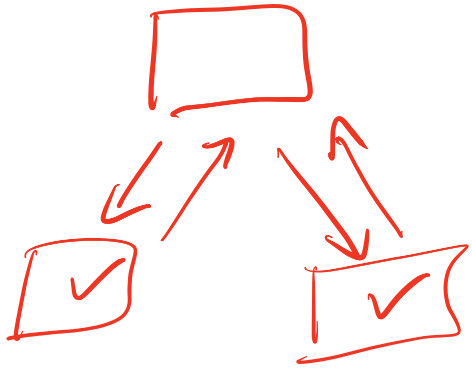
8am Monday 2/8

8am Tue 2/9

Recursion is...

Building solutions for *big problems* from solutions for *sub-problems*

- **Base case:** what is the *simplest version* of this problem and how do I solve it?
- **Inductive strategy:** how do I *break down* this problem into sub-problems?
- **Inductive case:** how do I solve the problem *given* the solutions for subproblems?



Lists

[1, 2, 3]

Lists aren't built-in! They are an *algebraic data type* like any other:

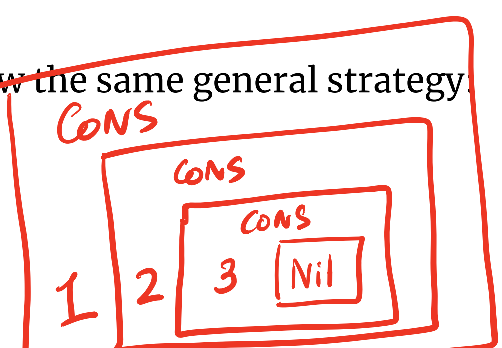
```
data List
  = Nil              -- ^ base constructor
  | Cons Int List    -- ^ inductive constructor
```

- List `[1, 2, 3]` is *represented* as `Cons 1 (Cons 2 (Cons 3 Nil))`
- Built-in list constructors `[]` and `(:)` are just fancy syntax for `Nil` and `Cons`

1 : (2 : (3 : []))

Cons 1 (Cons 2 (Cons 3 Nil))

Functions on lists follow the same general strategy:



```
length :: List -> Int
length Nil          = 0                 -- base case
length (Cons _ xs) = 1 + length xs   -- inductive case
```

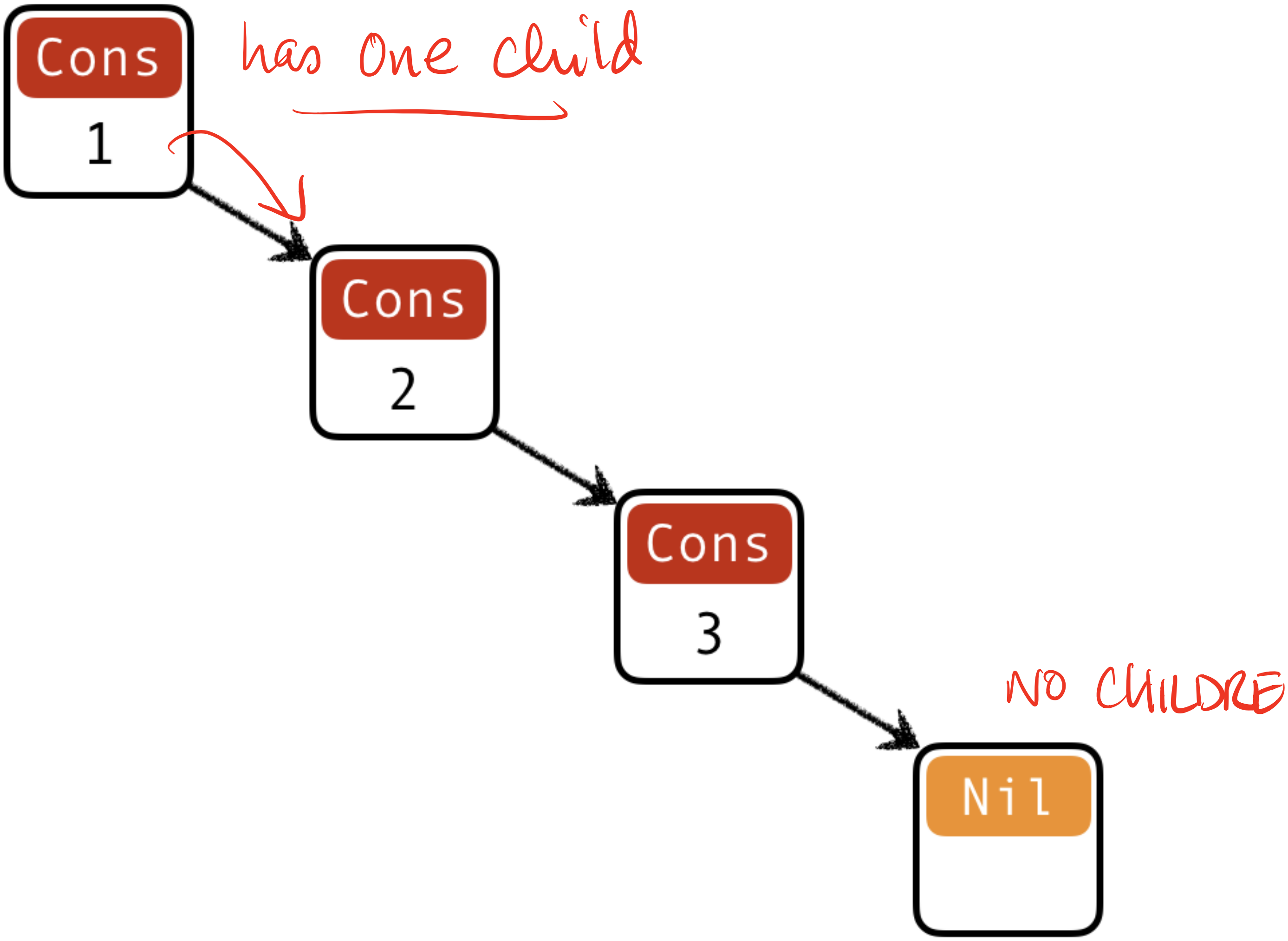
EXERCISE: Appending Lists

What is the right *inductive strategy* for appending two lists?

```
-- >>> append (Cons 1 (Cons 2 (Cons 3 Nil))) (Cons 4 (Cons 5 (Cons
6 Nil)))
-- (Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 (Cons 6 Nil))))))

append :: List -> List -> List
append xs ys = ??
```

Trees

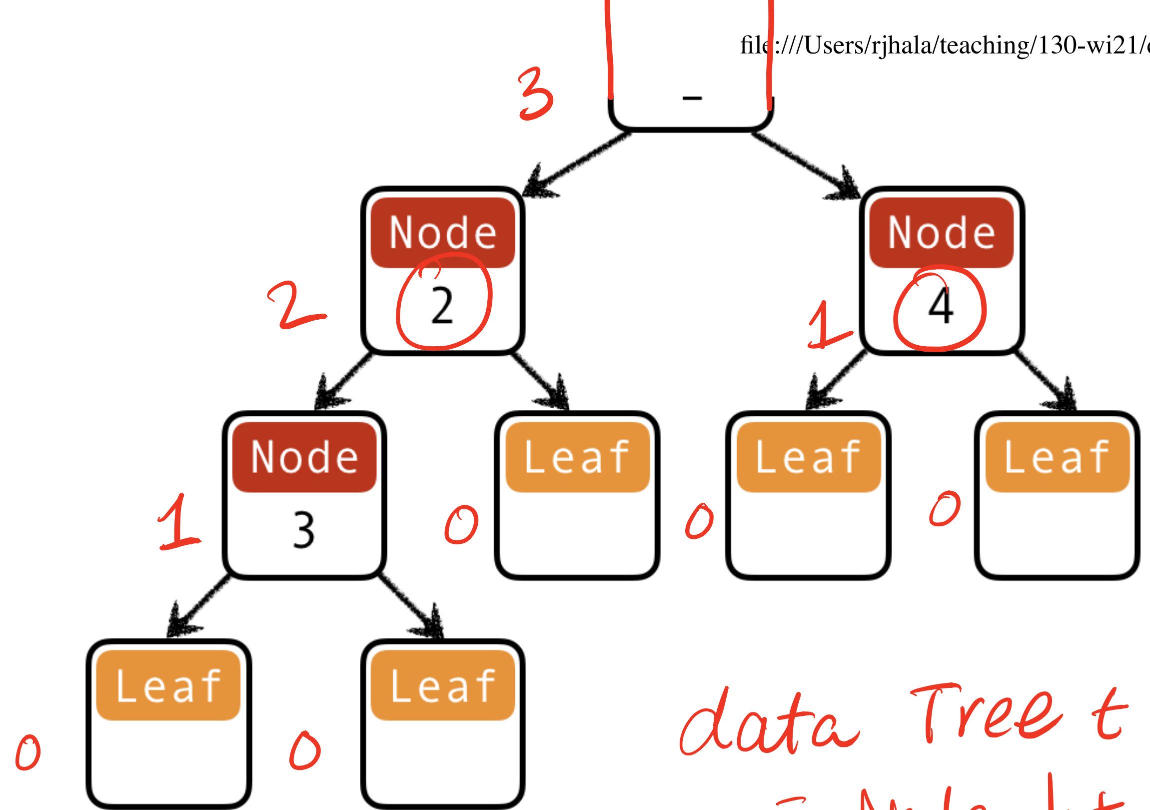
Lists are *unary trees* with elements stored in the nodes:



Lists are unary trees

```
data List = Nil | Cons Int List
```

How do we represent *binary trees* with elements stored in the nodes?

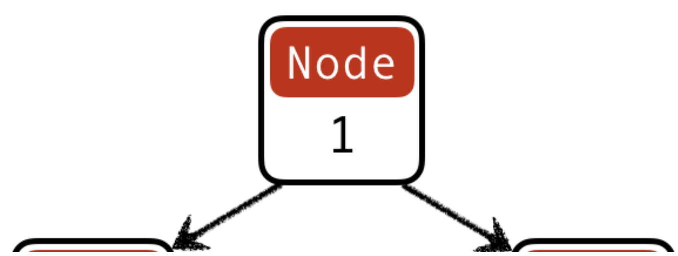


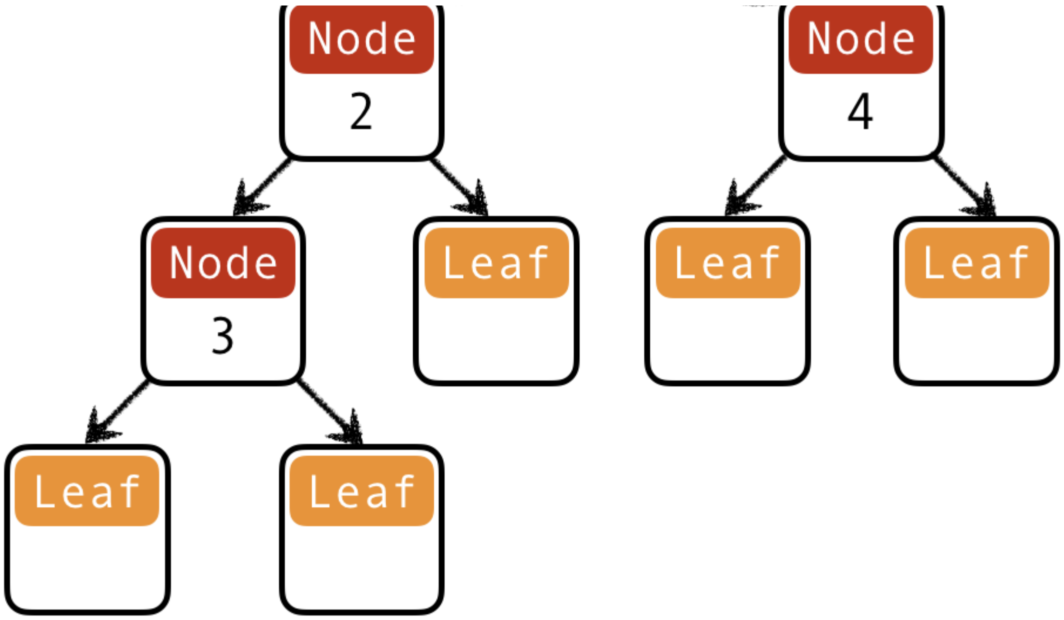
data Tree t
= Node Int (Tree t)(Tree t)
| Leaf

Binary trees with data at nodes

QUIZ: Binary trees I

What is a Haskell datatype for *binary trees* with elements stored in the nodes?





Binary trees with data at nodes

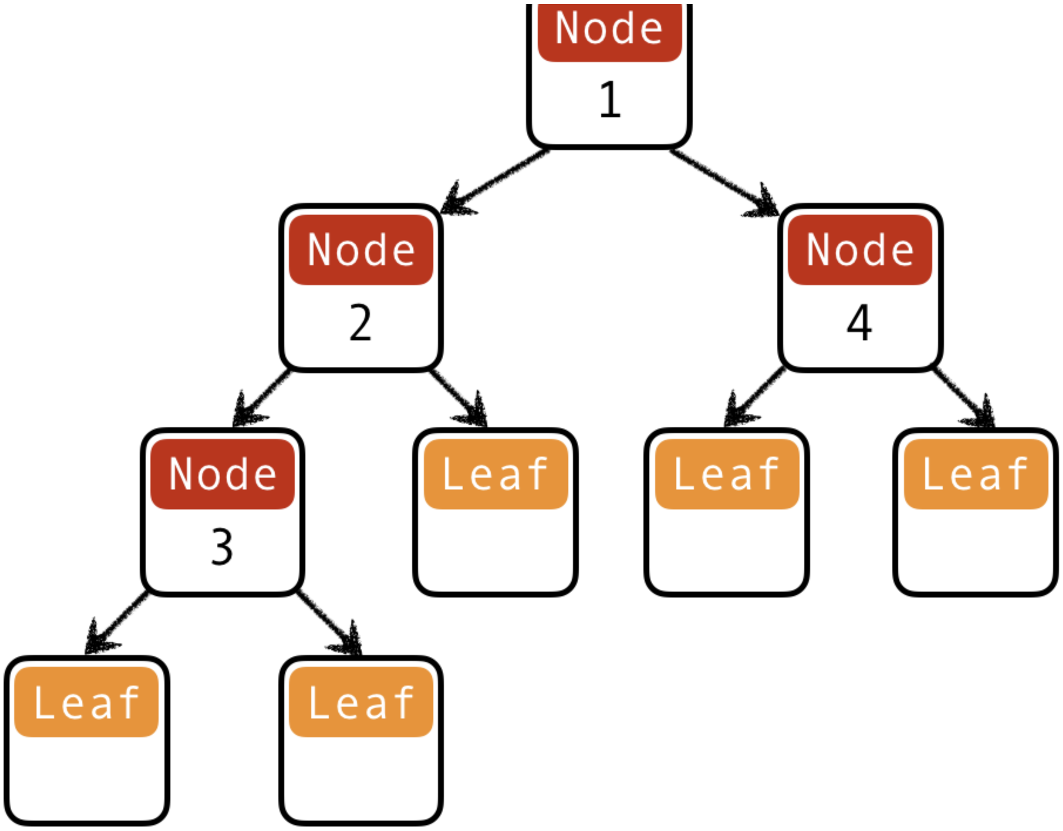
(A) `data Tree = Leaf | Node Int Tree`

(B) `data Tree = Leaf | Node Tree Tree`

(C) `data Tree = Leaf | Node Int Tree Tree`

(D) `data Tree = Leaf Int | Node Tree Tree`

(E) `data Tree = Leaf Int | Node Int Tree Tree`



Binary trees with data at nodes

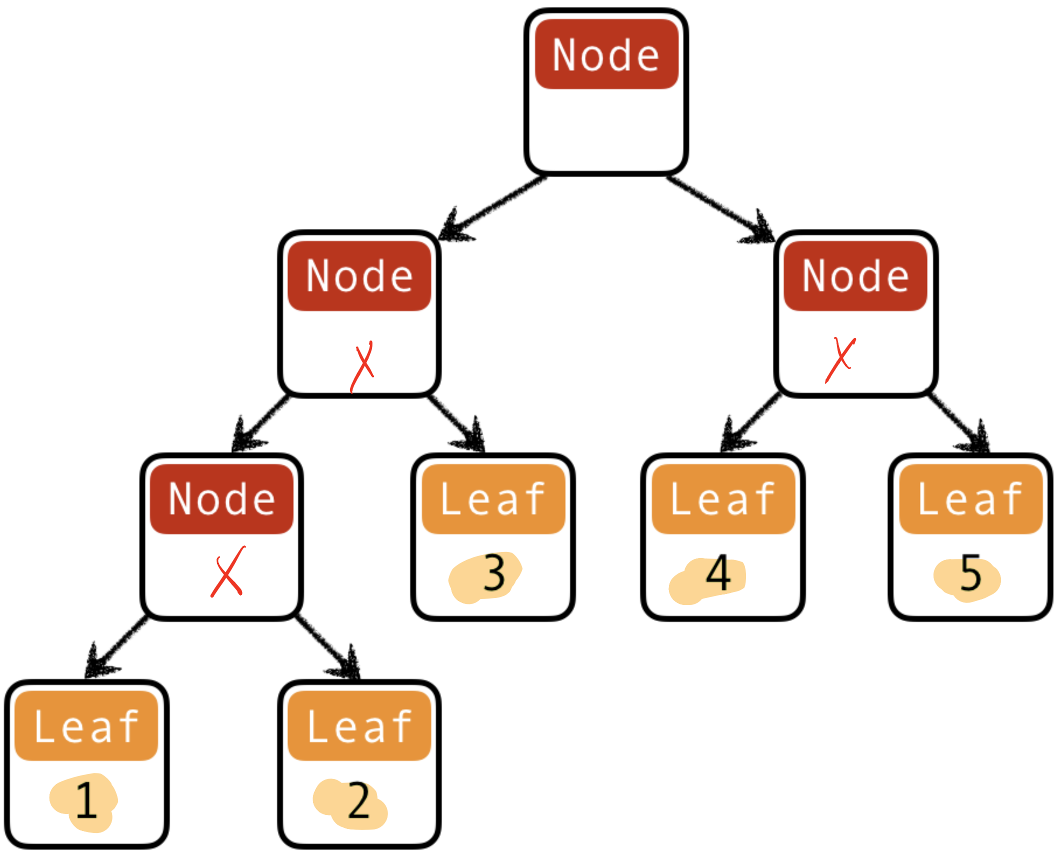
```
data Tree = Leaf | Node Int Tree Tree

t1234 = Node 1
          (Node 2 (Node 3 Leaf Leaf) Leaf)
          (Node 4 Leaf Leaf)
```

Functions on trees

```haskell
depth :: Tree -> Int
depth t = ??
```

QUIZ: Binary trees II

What is a Haskell datatype for *binary trees* with elements stored in the leaves?



Binary trees with data at leaves

```haskell
(A) data Tree = Leaf | Node Int Tree
```

(B) `data Tree = Leaf | Node Tree Tree`

(C) `data Tree = Leaf | Node Int Tree Tree`

(D) `data Tree = Leaf Int | Node Tree Tree` ✓

write depth max total

(E) `data Tree = Leaf Int | Node ~~Int~~ Tree Tree`

```haskell
data Tree = Leaf Int | Node Tree Tree

t12345 = Node
           (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
           (Node (Leaf 4) (Leaf 5))
```

treemax
depth
total

factor

Why use Recursion?

1. Often far simpler and cleaner than loops
    - But not always...
2. Structure often forced by recursive data
3. Forces you to factor code into reusable units (recursive functions)

map/reduce

Big-data

*Why **not** use Recursion?*

1. Slow
2. Can cause stack overflow

tail-recursion

⇓

fast-loop

Example: factorial

```haskell
fac :: Int -> Int
fac n
  | n <= 1     = 1
  | otherwise  = n * fac (n - 1)
```

not TR

```python
def fac(n):
  res = 1
  i = 1
  while (i <= n):
    res = res*i
    i = i+1
  return res
```

Lets see how `fac 4` is evaluated:

fac 100

```
<fac 4>
  ==> <4 * <fac 3>>                -- recursively call `fact 3`
  ==> <4 * <3 * <fac 2>>>          --   recursively call `fact 2`
  ==> <4 * <3 * <2 * <fac 1>>>>    --     recursively call `fact 1`
  ==> <4 * <3 * <2 * 1>>>          --     multiply 2 to result
  ==> <4 * <3 * 2>>                --   multiply 3 to result
  ==> <4 * 6>                      -- multiply 4 to result
  ==> 24
```

Each *function call* `<>` allocates a frame on the *call stack*

- expensive
- the stack has a finite size

Can we do recursion without allocating stack frames?