```
let p = Text "Hey there!"
in case p of
    PText _       -> 1
    PHeading _ _  -> 2
    PList _ _     -> 3
```

A. Syntax error

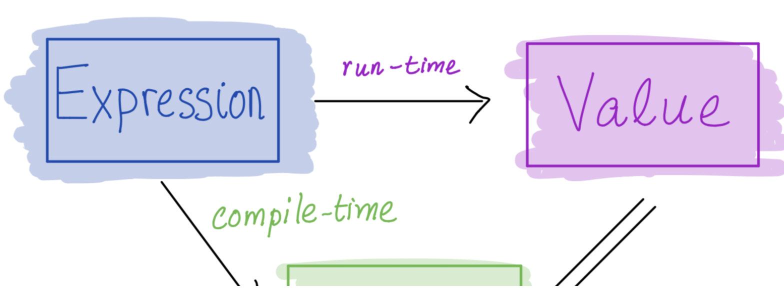
B. Type error

C. `Paragraph`

D. `Int`

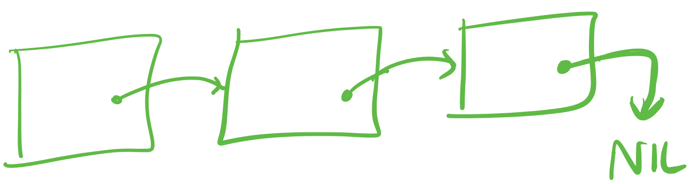
E. `Paragraph -> Int`

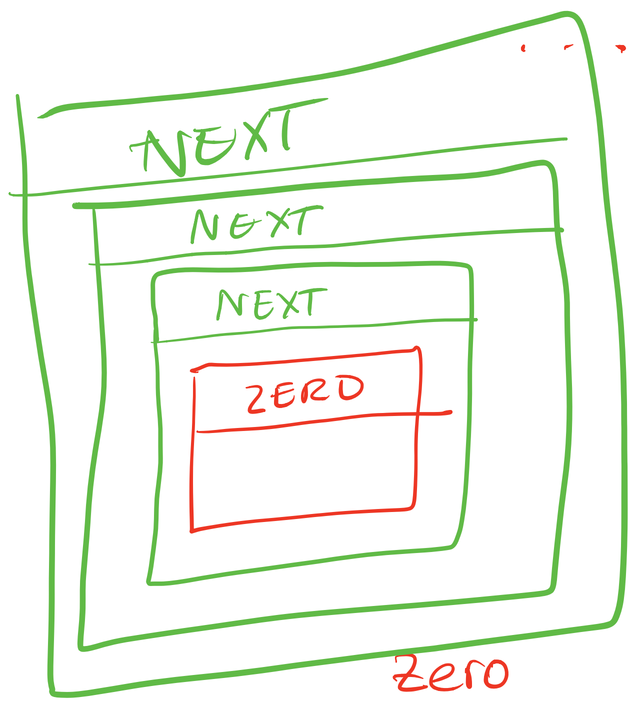
Building data types



Three key ways to build complex types/values:

1. **Product types (each-of)**: a value of `T` contains a value of `T1` *and* a value of `T2` **[done]**
    - Cartesian *product* of two sets: $v(T) = v(T1) \times v(T2)$
2. **Sum types (one-of)**: a value of `T` contains a value of `T1` *or* a value of `T2` **[done]**
    - Union (*sum*) of two sets: $v(T) = v(T1) \cup v(T2)$
3. **Recursive types**: a value of `T` contains a *sub-value* of the same type `T`





Recursive types

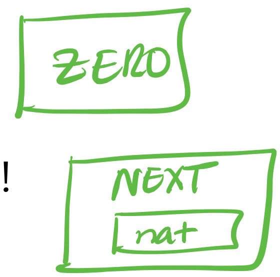
Let's define **natural numbers** from scratch:

```
data Nat = ???
```

```haskell
data Nat = Zero | Succ Nat
```

A `Nat` value is:

- either an *empty* box labeled `Zero`
- or a box labeled `Succ` with another `Nat` in it!



Some `Nat` values:

```haskell
Zero                     -- 0
Succ Zero                -- 1
Succ (Succ Zero)         -- 2
Succ (Succ (Succ Zero))  -- 3
...
```

Functions on recursive types

Recursive code mirrors recursive data

1. Recursive type as a parameter

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

Step 1: add a pattern per constructor

```
toInt :: Nat -> Int
toInt Zero     = ... -- base case
toInt (Succ n) = ... -- inductive case
                     -- (recursive call goes here)
```

Step 2: fill in base case:

```
toInt :: Nat -> Int
toInt Zero     = 0   -- base case
toInt (Succ n) = ... -- inductive case
                     -- (recursive call goes here)
```

Step 2: fill in inductive case using a recursive call:

```
toInt :: Nat -> Int
toInt Zero     = 0           -- base case
toInt (Succ n) = 1 + toInt n -- inductive case
```

data Nat = Zero | Succ Nat

QUIZ

aka "Next"

What does this evaluate to?

```
let foo i = if i <= 0 then Zero else Succ (foo (i - 1))
in foo 2
```

A. Syntax error

B. Type error

C. `2`

D. `Succ Zero`

E. `Succ (Succ Zero)`

2. Recursive type as a result

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

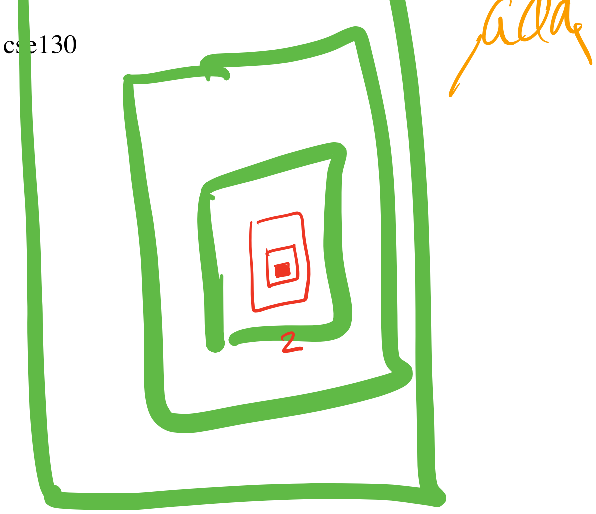
```
fromInt :: Int -> Nat
fromInt n
  | n <= 0    = Zero                     -- base case
  | otherwise = Succ (fromInt (n - 1))  -- inductive case
                                         -- (recursive call goes her
e)
```

EXERCISE: Putting the two together

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor

add :: Nat -> Nat -> Nat
add n m = ???

sub :: Nat -> Nat -> Nat
sub n m = ???
```



EXERCISE: Putting the two together

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

```
add :: Nat -> Nat -> Nat
add n m = ???
```

```haskell
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

```haskell
add :: Nat -> Nat -> Nat
add Zero     m = ???            -- base case
add (Succ n) m = ???            -- inductive case
```

EXERCISE: Putting the two together

```haskell
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor

sub :: Nat -> Nat -> Nat
sub n m = ???
```

```haskell
sub :: Nat -> Nat -> Nat
sub n        Zero     = ???     -- base case 1
sub Zero     _        = ???     -- base case 2
sub (Succ n) (Succ m) = ???     -- inductive case
```

Lesson: Recursive code mirrors recursive data

- Which of **multiple** arguments should you recurse on?
- Key: Pick the right **inductive strategy**!

(easiest if there is a *single* argument of course…)

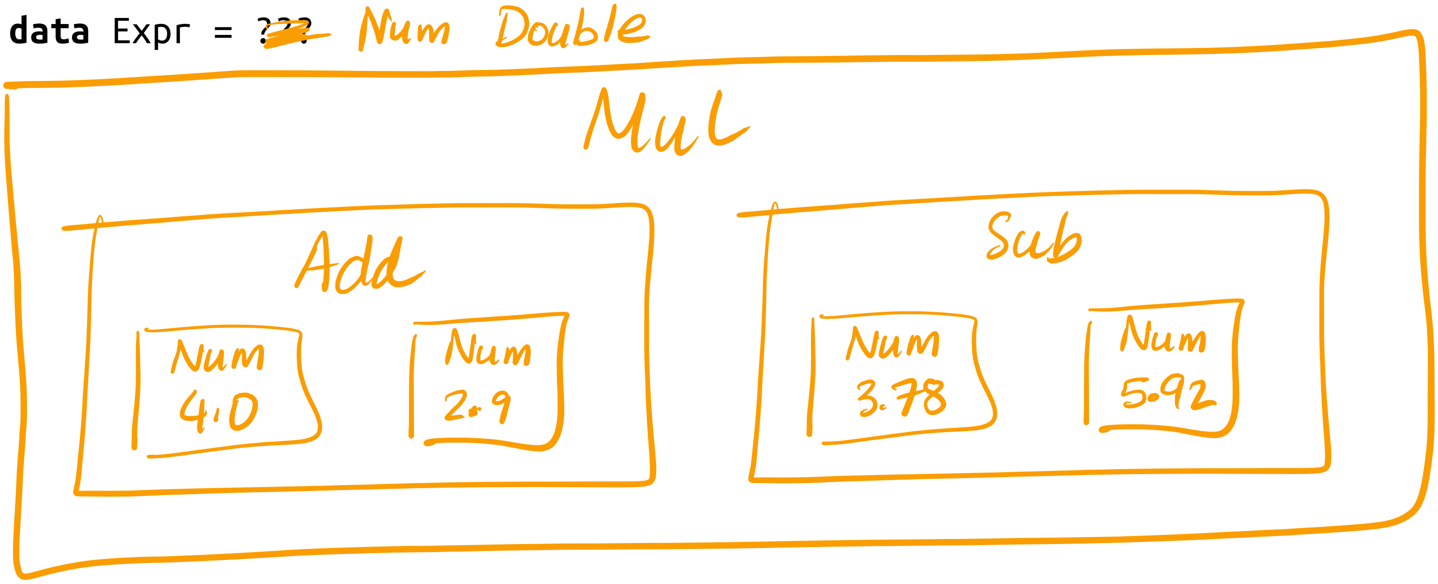
Example: Calculator

I want to implement an arithmetic calculator to evaluate expressions like:

- `4.0 + 2.9`
- `3.78 - 5.92`
- `(4.0 + 2.9) * (3.78 - 5.92)`

What is a Haskell datatype to *represent* these expressions?

```
data Expr = ???
```




```
data Expr = Num Float
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
```

We can represent expressions as