Pure + Lazy

No mutation

Exp not evaluated until needed

Static Typing: Every exp has a type known at compile time.

If you call func with wrong type, the compiler rejects.

Type Inference: You don't have to write types everywhere. The compiler infers the most general type that makes code type-check.

```haskell
square x = x * x
```

"has type"

```haskell
:t square
-- square :: Num a => a -> a
```

square works $\forall a \in$ Num (Int, Integer, Double) id :: a → a

Give me any type 'a'. I take an 'a' and return an 'a' (same type)

$a \to a \to a \equiv a \to (a \to a)$

Give me an 'a', and I'll return a func $(a \to a)$ that takes 2nd 'a'

Eq a ⇒ … ∀ a that supports equality.

(==), (/=) :: Eq a ⇒ a → a → Bool

If your type lacks Eq, you can't use (==) on it.

All Haskell keywords are lowercase

data Either a b = Left a | Right b

keyword

type vars

data constructor

type Constructor

Constructors are func

For 'Either a b'

Left :: a -> Either a b

Right :: b -> Either a b

(String = [Char])

e.g. ['h', 'i'] :: [Char] ≡ "hi" :: String

We build list with cons operator ':'

Maybe => Might be missing

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

To produce 'Maybe a', we must use Nothing / Just __

Both branches must have same type.
Here, Maybe a, Nothing, Just are data constructors for that type

Just 'a' :: Maybe Char
[∵ Just :: t → Maybe Char]
Here, 'a' :: Char

Nothing :: Maybe a
polymorphic – works ∀ a
Context decides which 'Maybe a' you meant. (e.g. Maybe Int, Maybe String)

Everything to the right of :: must be a type, not value. Types use type constructors like Maybe, Either, [ ]
And other types (Bool, Int)

True :: Bool
Just True :: Maybe Bool
Just (Just True) :: Maybe (Maybe Bool)

Just [True, False] :: Maybe [Bool]

3 Ways to define a "point"

```haskell
type Point1 = (Float, Float)
-- Alias (just a nickname)
```

type = alias
No new type is created
Point1 and (Float, Float) are
identical to type checker

```haskell
p1 :: Point1
p1 = (1.0, 2.0)
len :: (Float, Float) -> Float
len (x,y) = sqrt (x * x + y * y)
len p1
```

```haskell
newtype Point2 = Point2 (Float,
                         Float)
-- Wrapper
```

(brand-new type, 1 field)
Must wrap/unwrap with its
constructor name.

```haskell
p2 :: Point2
p2 = Point2 (1.0, 2.0)
```

len p2 ✗ type error
[Point2 ≠ (Float, Float)]

let (Point2 t) = p2 in len t ✓

keyword that introduces local binding
let <pattern> = <exp> in <body>

(Point2 t) ⇒ a pattern that matches a value built with data constructor Point2 and binds to single field t.
Data constructors are real func
Point2 :: (Float, Float) → Point2

This method is type safe and we can give its own instance.

```haskell
data Point3 = Point3 Float Float
-- Wrapper (brand-new type, 2 Fields)
```

Allows multiple fields/constructor

```haskell
p3 :: Point3
p3 = Point3 1.0 2.0
```

len p3 X

```haskell
let (Point3 x y) = p3 in len (x, y)
```

```haskell
Point3 :: Float -> Float -> Point3
=> Point3 :: Float -> (Float -> Point3)
```

Supply 1 arg and you get back func waiting for 2nd

```haskell
Point3 3.2 :: Float -> Point3
```

A func that still needs 1 float

```haskell
Point3 3.2 4.5 :: Point3
```

Now fully applied; we have a value

Given,

$$map :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$

Give me func from $\alpha \to \beta$ and a list of α, and I'll give you list of β

In our case the func $\alpha \to \beta$ would be 'Point3 0.0' which has a type 'Float → Point3'

$\Rightarrow$ Here, α = Float, β = Point3

$$\Rightarrow map :: (Float \to Point3) \to [Float] \to [Point3]$$

Map applies our func to each element and collects the results.

```
f :: a -> b
```
[Give me value of type 'a', and I return value of type 'b']

```
f :: C a => a -> b
```
[∀ 'a' that satisfies class C, give me 'a', I return 'b']

```
len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

```
sum' :: Num a => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

```
elem' :: Eq a => a -> [a] -> Bool
elem' x [] = False
elem' x (y:ys) = (x == y) || elem' x ys
```