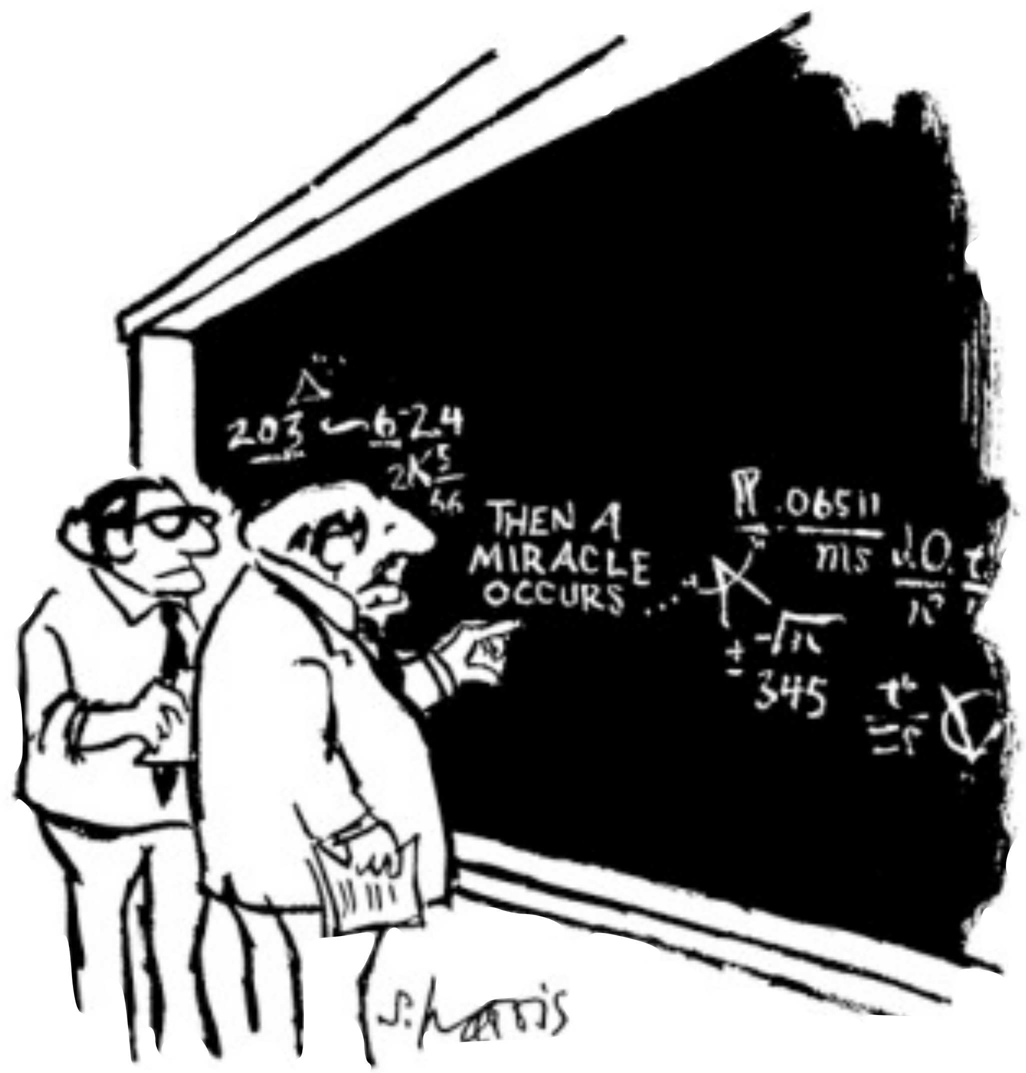
What is FP? and How to Do It!

(an expedited refresher w/ bonus goodies)



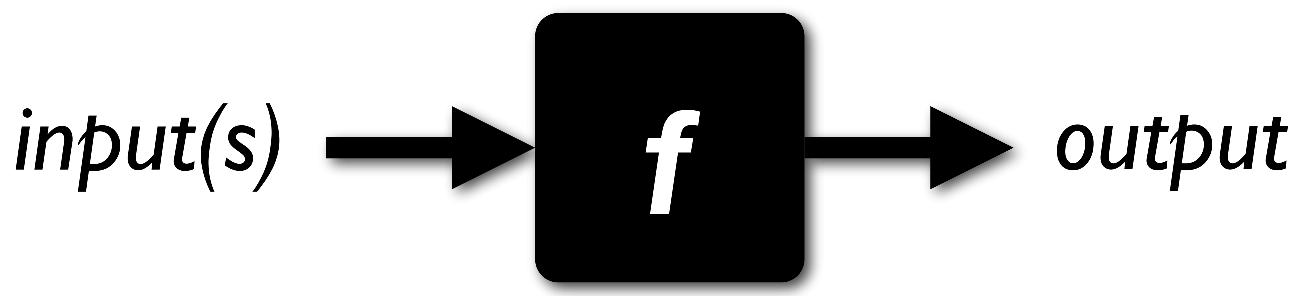
"I think you should be more explicit here in step two."

Sidney Harris, 1977

Outline

- **The essence of functional programming**
- FP workflow and type-directed programming
- A closer look at types – parametricity

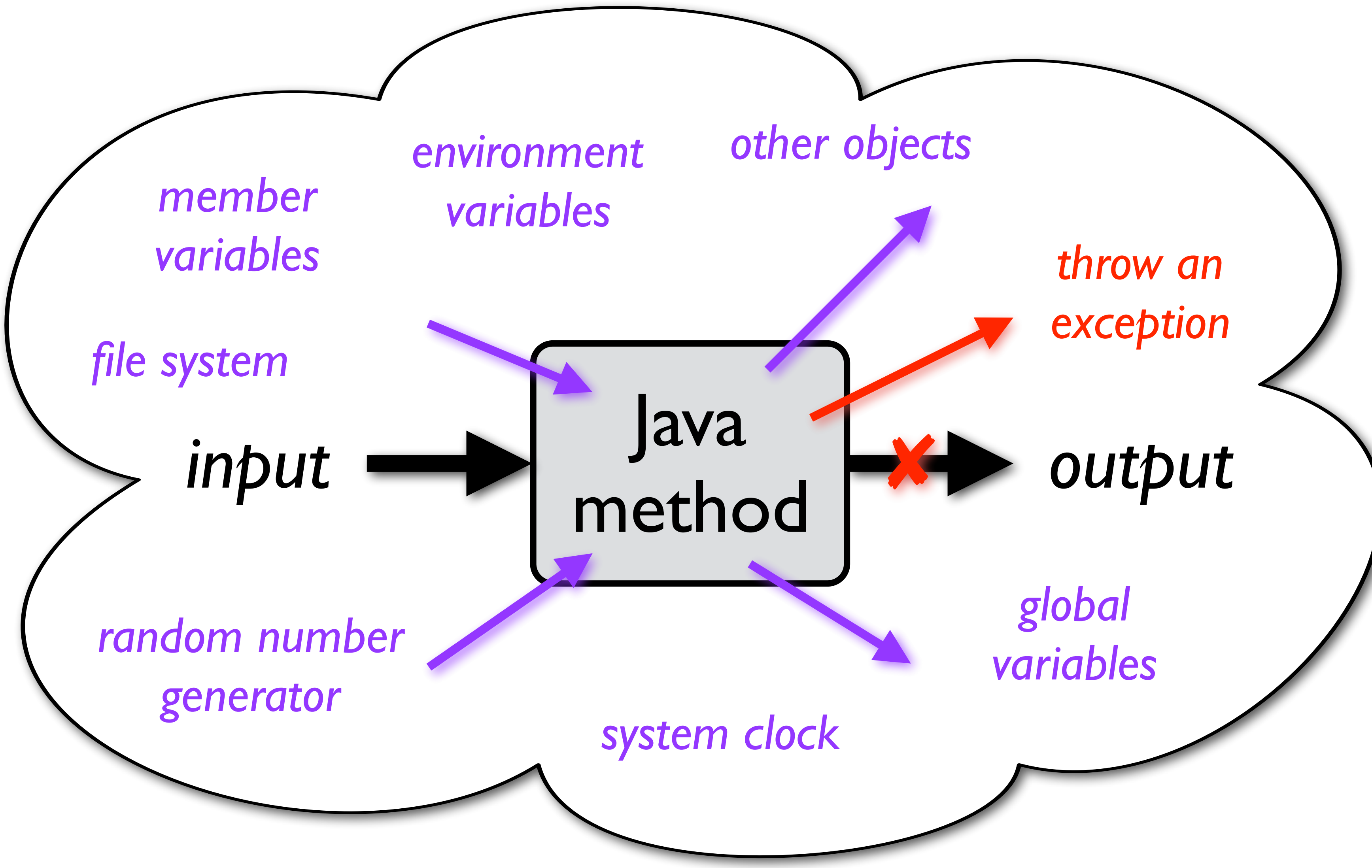
Functions are pure!



In Haskell, all functions are *pure*:

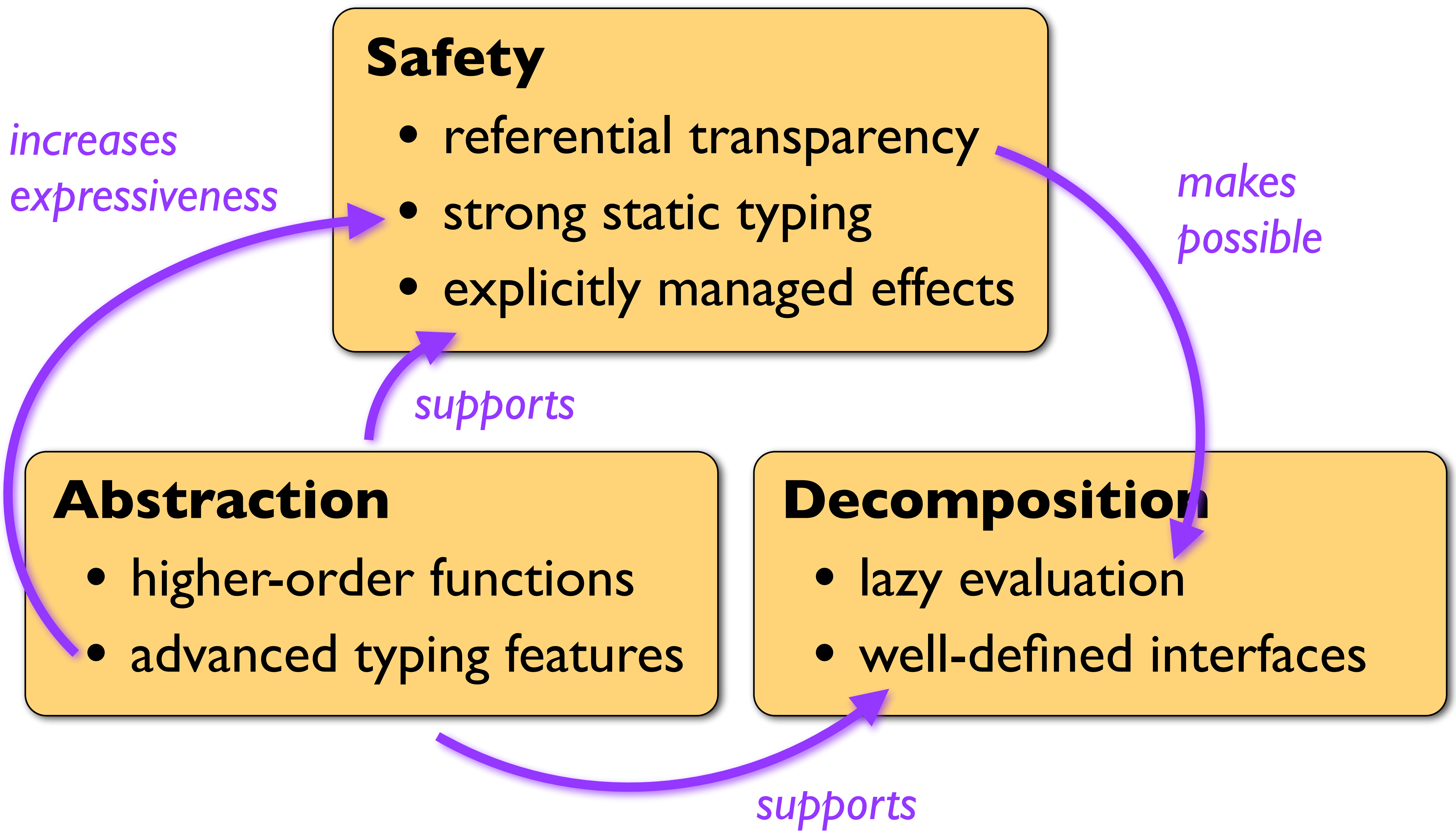
- always return the same output for the same inputs
- don't do anything else – no "side effects"

Procedures/methods aren't functions



- output depends on environment
- may perform arbitrary side effects

Guiding principles of FP



Referential transparency

a.k.a. ***referent***

An expression can be replaced by its **value** without changing the overall program behavior

```
\x -> crunch [5,6,7] + x
       ⇒      \x -> 3 + x
```

what if `crunch` *was a Python function?*

Corollary: an expression can be replaced by **any expression** with the same value without changing program behavior

Supports decomposing a problem into parts and "equational reasoning"

Equational reasoning

Computation is just substitution!

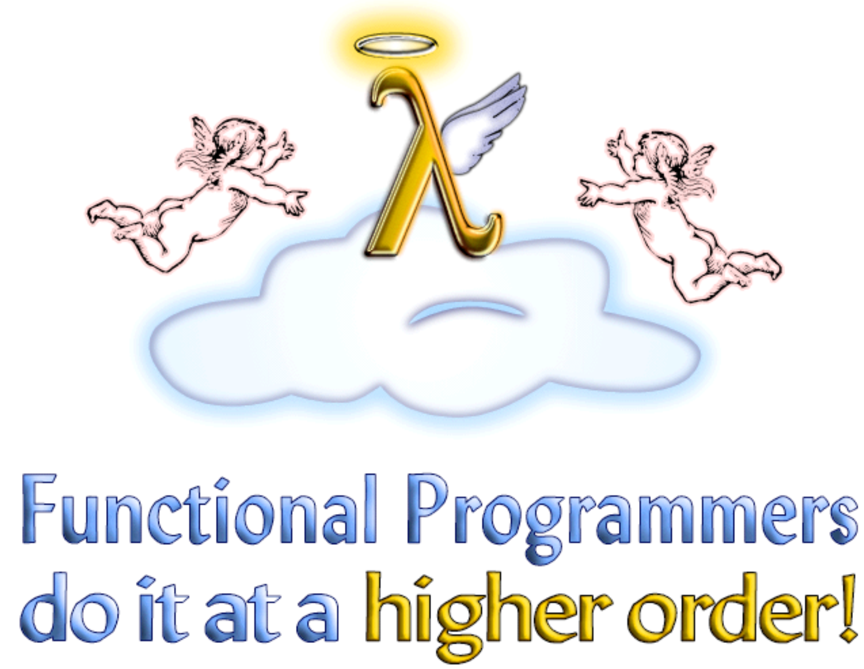
```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

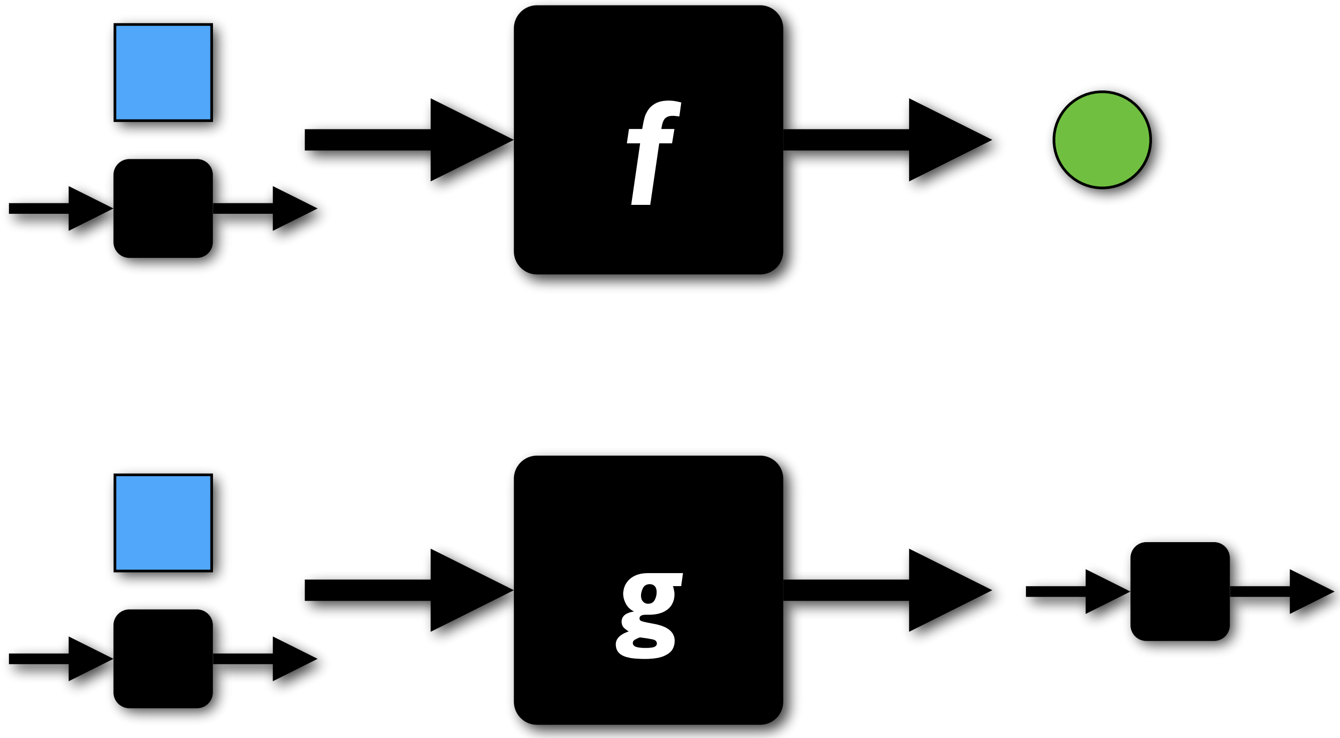
equations

```
   sum [2,3,4]
⇒ sum (2:(3:(4:[])))
⇒ 2 + sum (3:(4:[]))
⇒ 2 + 3 + sum (4:[])
⇒ 2 + 3 + 4 + sum []
⇒ 2 + 3 + 4 + 0
⇒ 9
```

Very useful when refactoring (later)

Higher-order functions





Examples:

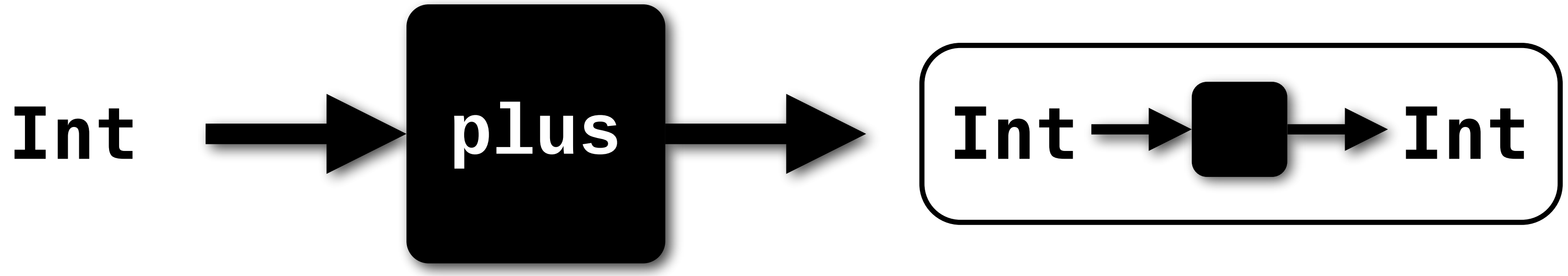
```
filter :: (a -> Bool) -> [a] -> [a]
map    :: (a -> b) -> [a] -> [b]
(.)    :: (b -> c) -> (a -> b) -> a -> c
```

Currying / partial application

In Haskell, functions that take multiple arguments are *technically* implicitly higher-order

Haskell Curry

```
plus :: Int -> Int -> Int
```




```
increment :: Int -> Int
increment = plus 1
```

Uncurried version: *"cannot" be partially applied!*

```
plus :: (Int,Int) -> Int
```

Lazy evaluation

Expressions are reduced:

- only when needed
- at most once

Supports:

- infinite data structures
- efficient and simple **separation of concerns** (decomposition)

Efficiently implemented using graph-reduction (later)

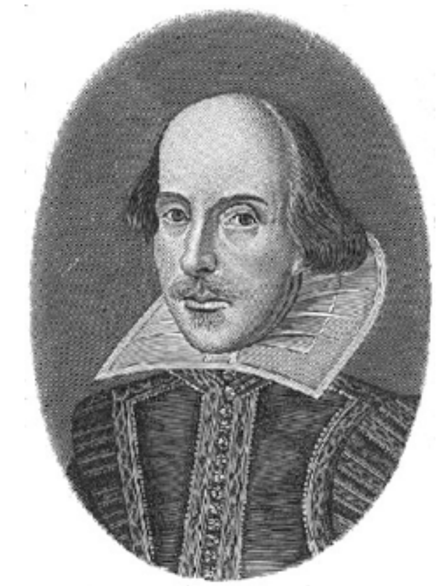
(Lazy.hs, NQueens.hs)

John Hughes, Why Functional Programming Matters, 1989

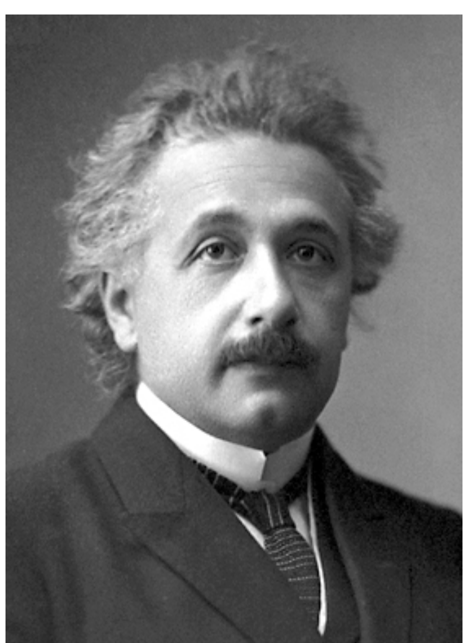
Outline

- The essence of functional programming
- **FP workflow and type-directed programming**
- A closer look at types – parametricity

Striving for elegance



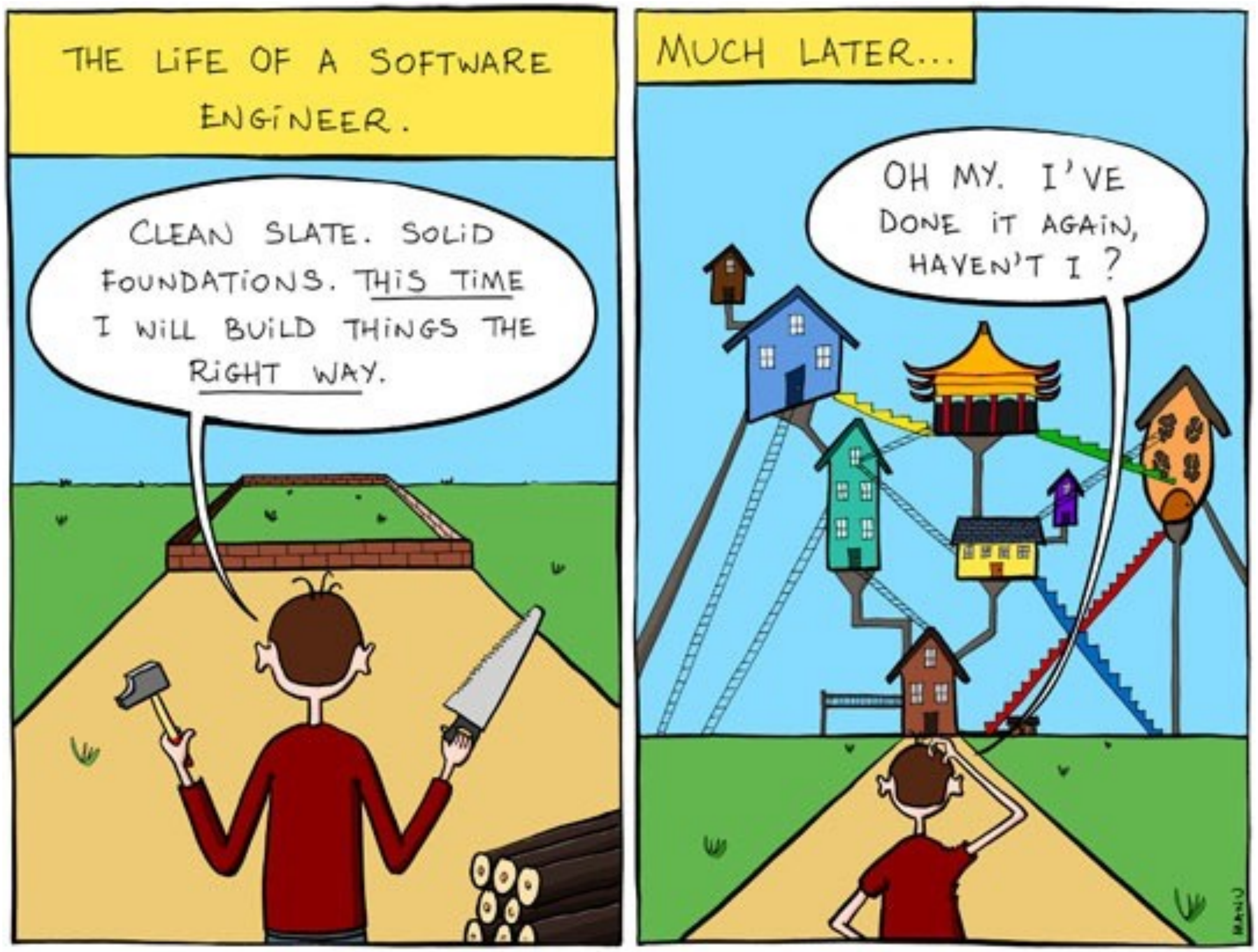
the quality of being pleasingly ingenious and simple; neatness
— New Oxford American Dictionary



the beauty of an idea characterized by minimalism and intuitiveness while preserving exactness and precision
— Wiktionary

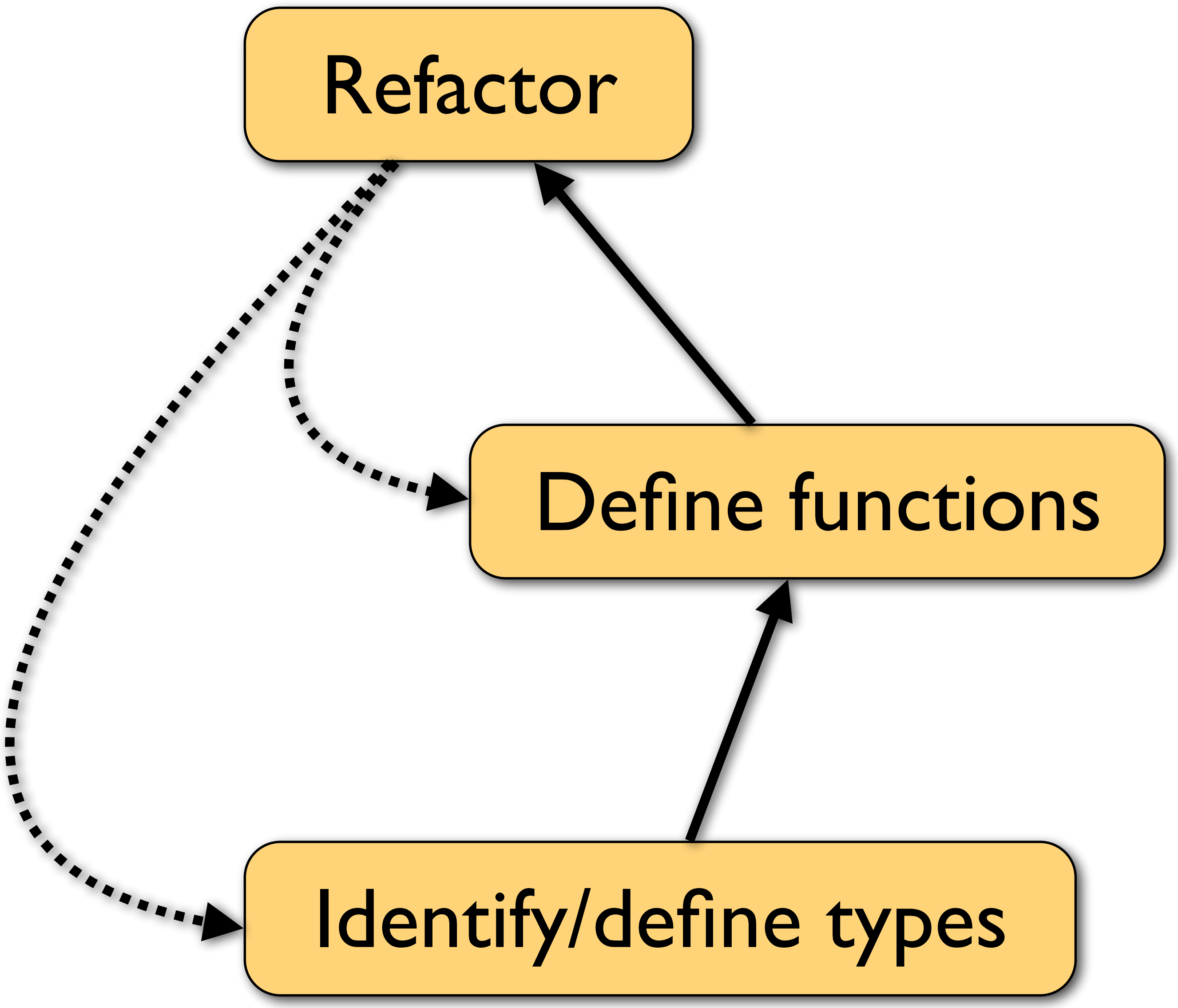
"obsessive compulsive refactoring disorder"

Striving for elegance

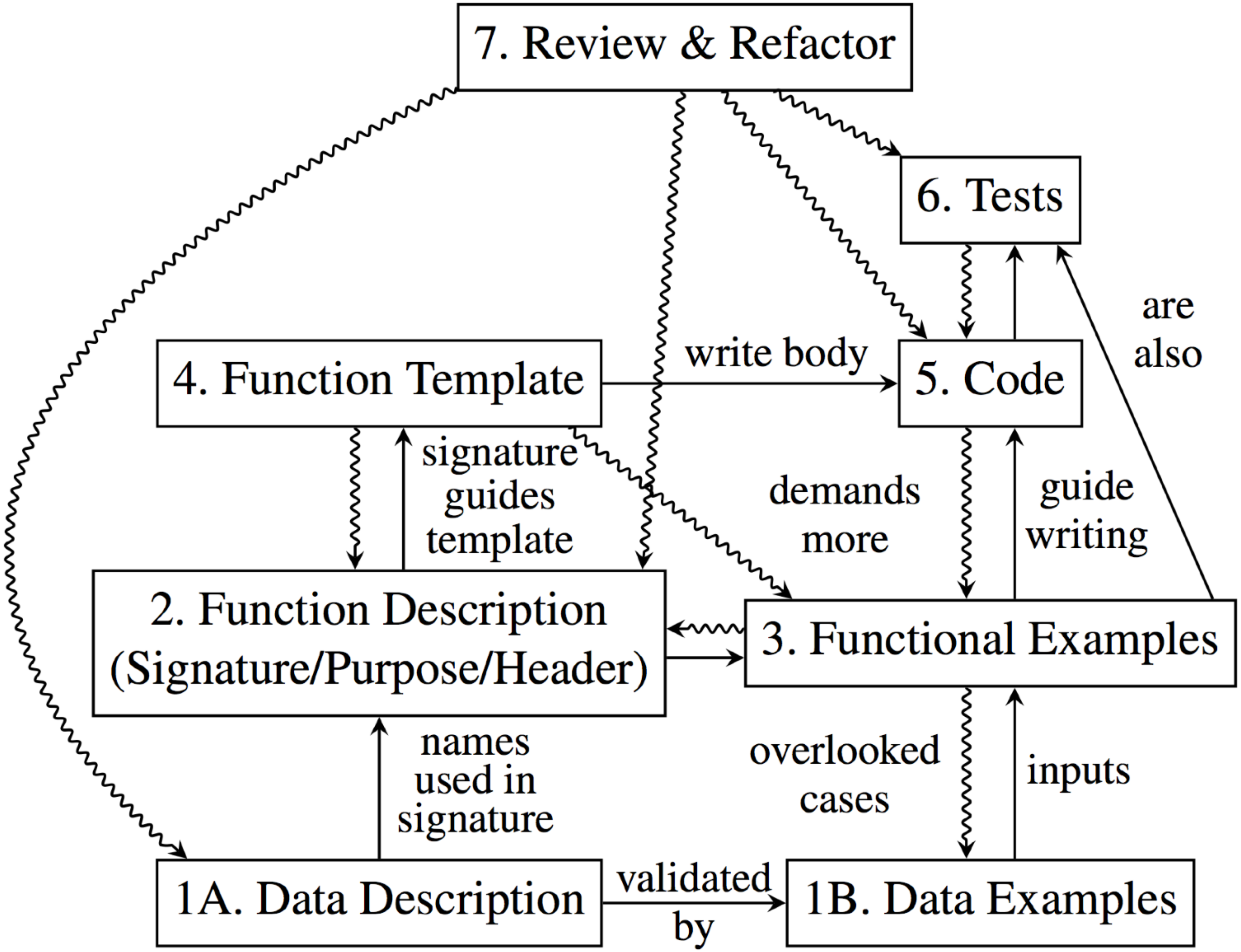


don't stop here!

FP workflow (simple)



FP workflow (detailed)



Ramsey, On Teaching *How to Design Programs*, ICFP'14

Anatomy of a data type

multiple cases = *"sum type"*

multiple arguments = *"product type"*

refers to self = *"recursive type"*

not decomposable = *"atomic type"*

type name *type parameter*

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

cases

data constructor *arguments*

```
exTree :: Tree Int
exTree = Node (Leaf 2) (Node (Leaf 3) (Leaf 4))
```

Tools for defining functions

Recursion and other functions:

```
sum :: [Int] -> Int
sum xs = if null xs then 0
         else head xs + sum (tail xs)
```

Pattern matching:

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

(1) case analysis

(2) decomposition

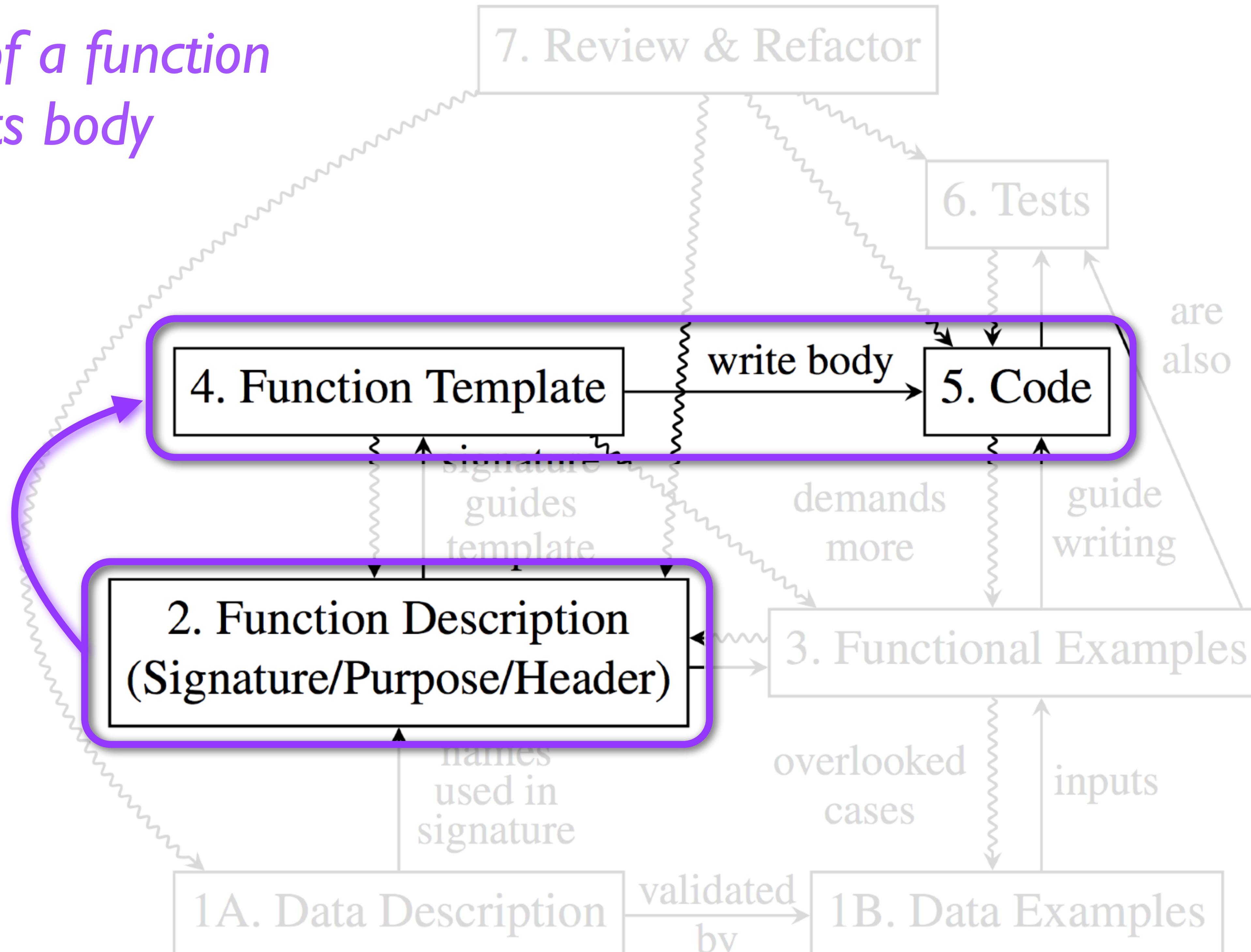
Higher-order functions

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

no recursion or variables needed!

What is type-directed programming?

Use the type of a function to help write its body



If your argument is . . .

sometimes match against literals

atomic type	→	apply functions to it
function type	→	apply it
sum type	→	use case analysis on it
product type	→	decompose it

often pattern matching

. . . *and repeat as necessary*

Atomic type: apply functions to it

or pattern match against literals

Examples: **`Int`**, **`Bool`**, **`Float`**

```
isZero :: Int -> Bool

isZero n = f n
isZero n = (== 0) n
isZero n = n == 0

isZero 0 = True
isZero _ = False
```

Sum type: use case analysis on it

often by pattern matching

Example: `data Maybe a = Nothing | Just a`

```
showValue :: Maybe Int -> String
```

Option 1: pattern matching

```
showValue Nothing  = "ERROR"
showValue (Just i) = show i
```

Option 2: case analysis function

```
maybe :: b -> (a -> b) -> Maybe a -> b

showValue val = maybe "ERROR" show val
```

Product type: decompose it

often by pattern matching

Example: `type Point = (Float,Float)`

```
moveR :: Float -> Point -> Point
```

Option 1: pattern matching

return type is a product, so we must ***construct*** *it*

```
moveR s (x,y) = (x+s, y)
```

Option 2: destructor functions

```
moveR s p = (fst p + s, snd p)
```

Function type: apply it

Example: `a -> b`

```
pipe :: (a -> b) -> (b -> c) -> a -> c

pipe f g a = ...
pipe f g a = ... f a ...
pipe f g a = g (f a)
```

Outline

- The essence of functional programming
- FP workflow and type-directed programming
- **A closer look at types – parametricity**

Parametricity – what's in a type?

If I give you *only the* ***type*** of a function …

- can you implement it?
- what else can you say about it?

Exercise: implement each of the following functions

$$f_1 :: a \to b \to a$$
$$f_2 :: a \to (a \to b) \to b$$
$$f_3 :: a \to a \to (a \to b) \to b$$
$$f_4 :: a \to b$$

Theorems for free!

```
sum    :: [Int] -> Int
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

Consider this function:

```
glob :: [a] -> [a]
```

What can it **potentially** do? *rearrange, copy, delete*

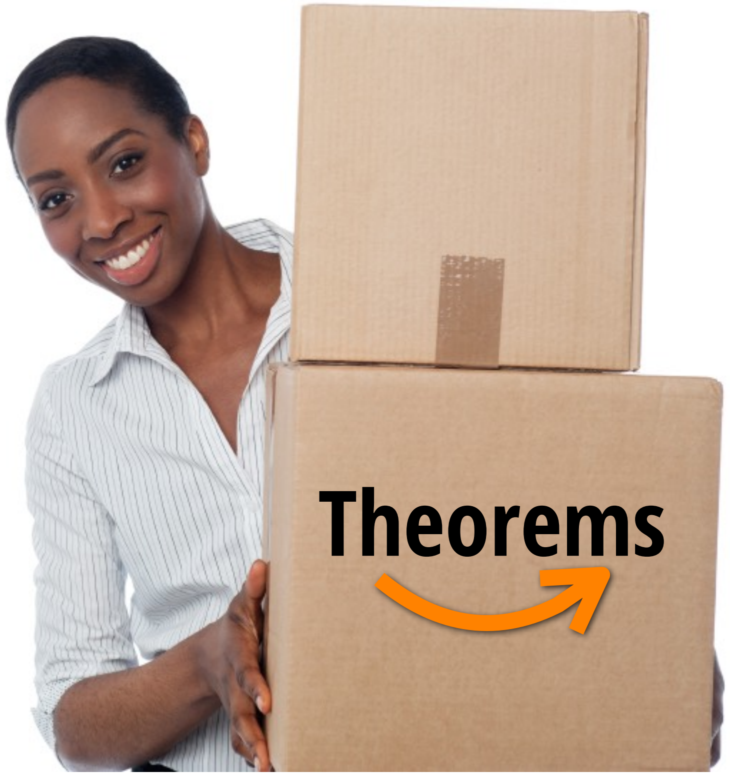
What can it **definitely not** do? *anything that depends on the **values** in list!*

Which of the following theorems are true?

```
     sum . glob  <=>  glob . sum       ✗
   map f . glob  <=>  glob . map f     ✓
filter f . glob  <=>  glob . filter f  ✗
```

Phil Wadler, Theorems for free! ICFP'89

Deliver some free theorems!




```
head   :: [a] -> a
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

+ map is structure-preserving

```
     head . map f   <=>   f . head

filter p . map f   <=>   map f . filter (p . f)

    map g . map f   <=>   map (g . f)
```