Bottling Computation Patterns

Polymorphism and HOFs are the Secret Sauce

Refactor arbitrary *repeated* code patterns ...

... into precisely *specified* and *reusable* **functions**

EXERCISE: Iteration

Write a function that *squares* a list of `Int`

```
squares :: [Int] -> [Int]
squares ns = ???
```

When you are done you should see

```
>>> squares [1,2,3,4,5]
[1,4,9,16,25]
```

Pattern: Iteration

Next, lets write a function that converts a `String` to uppercase.

```
>>> shout "hello"
"HELLO"
```

Recall that in Haskell, a `String` is just a `[Char]`.

```
shout :: [Char] -> [Char]
shout = ???
```

Hoogle (http://haskell.org/hoogle) to see how to transform an individual `Char`

Iteration

Common strategy: *iteratively* transform *each element* of input list

Like humans and monkeys, `shout` and `squares` share 93% of their DNA (http://www.livescience.com/health/070412_rhesus_monkeys.html)

Super common *computation pattern*!

Abstract Iteration "Pattern" into Function

Remember D.R.Y. (Don't repeat yourself)

Step 1 Rename all variables to remove accidental *differences*

```
-- rename 'squares' to 'foo'
foo []     = []
foo (x:xs) = (x * x)     : foo xs

-- rename 'shout' to 'foo'
foo []     = []
foo (x:xs) = (toUpper x) : foo xs
```

Step 2 Identify what is *different*

- In `squares` we *transform* `x` to `x * x`
- In `shout` we *transform* `x` to `Data.Char.toUpper x`

Step 3 Make *differences* a parameter

- Make *transform* a parameter `f`

```
foo f []     = []
foo f (x:xs) = (f x) : foo f xs
```

Done We have *bottled* the computation pattern as `foo` (aka `map`)

```
map f []     = []
map f (x:xs) = (f x) : map f xs
```

`map` bottles the common pattern of iteratively transforming a list:

Fairy In a Bottle

QUIZ

aka folter

doTwice f x = f (f x)

What is the type of `map` ?

```
map :: ???
map f []     = []
map f (x:xs) = (f x) : map f xs
```

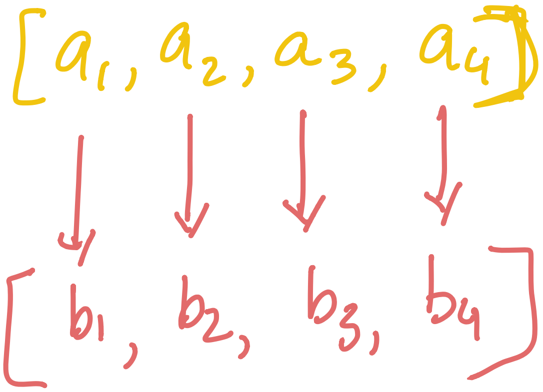
A. `(Int -> Int) -> [Int] -> [Int]`

[✓ **B.** `(a -> a) -> [a] -> [a]`]

C. `[a] -> [b]`

[✓ **D.** `(a -> b) -> [a] -> [b]`]

E. `(a -> b) -> [a] -> [a]`



The type precisely describes `map`

```
>>> :type map
map :: (a -> b) -> [a] -> [b]
```

That is, `map` takes two inputs

- a *transformer* of type `a -> b`
- a *list* of values `[a]`

and it returns as output

- a list of values `[b]`

that can only come by applying `f` to each element of the input list.

Reusing the Pattern

Lets reuse the pattern by *instantiating* the transformer

shout

```
-- OLD with recursion
shout :: [Char] -> [Char]
shout []     = []
shout (x:xs) = Char.toUpper x : shout xs

-- NEW with map
shout :: [Char] -> [Char]
shout xs = map (???) xs
```

squares

```
-- OLD with recursion
squares :: [Int] -> [Int]
squares []     = []
squares (x:xs) = (x * x) : squares xs

-- NEW with map
squares :: [Int] -> [Int]
squares xs = map (???) xs
```

EXERCISE

Suppose I have the following type

```
type Score = (Int, Int) -- pair of scores for Hw0, Hw1
```

Use `map` to write a function

```
total :: [Score] -> [Int]
total xs = map (???) xs
```

such that

```
>>> total [(10, 20), (15, 5), (21, 22), (14, 16)]
[30, 20, 43, 30]
```

The Case of the Missing Parameter

Note that we can write `shout` like this

```
shout :: [Char] -> [Char]
shout = map Char.toUpper
```

Huh. No parameters? Can someone explain?

The Case of the Missing Parameter

In Haskell, the following all mean the same thing

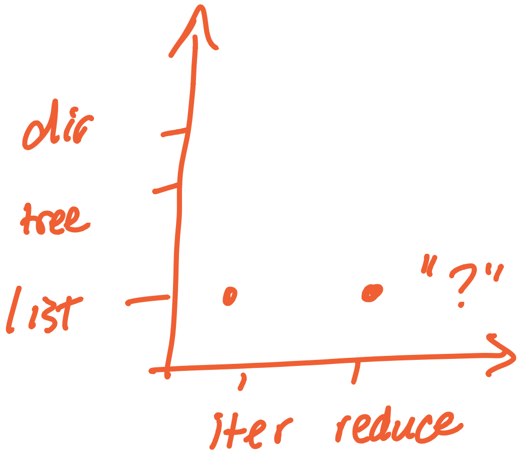
Suppose we define a function

```
add :: Int -> Int -> Int
add x y = x + y
```

Now the following all *mean the same thing*

```
plus x y = add x y
plus x   = add x
plus     = add
```

Why? *equational reasoning!* In general



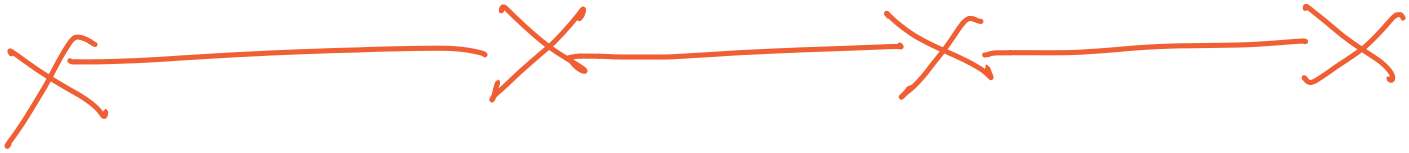
```
foo x = e x

-- is equivalent to

foo   = e
```

as long as `x` doesn't appear in `e`.

Thus, to save some typing, we *omit* the extra parameter. patterns

HOFs & Bottling Patterns

Hello World! (10)



Pattern: Reduction

Computation patterns are *everywhere* lets revisit our old `sumList`

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

Next, a function that *concatenates* the `String` s in a list

```
catList :: [String] -> String
catList []     = ""
catList (x:xs) = x ++ (catList xs)
```

Lets spot the pattern!

Step 1 Rename

```
foo []     = 0
foo (x:xs) = x + foo xs

foo []     = ""
foo (x:xs) = x ++ foo xs
```

Step 2 Identify what is *different*

1. ???
2. ???

Step 3 Make *differences* a parameter

```
foo p1 p2 []     = ???
foo p1 p2 (x:xs) = ???
```

EXERCISE: Reduction/Folding

This pattern is commonly called *reducing* or *folding*

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op base []     = base
foldr op base (x:xs) = op x (foldr op base xs)
```

Can you figure out how `sumList` and `catList` are just *instances* of `foldr` ?

sumList = foldr (+) 0

catList = foldr (++) ""

```
sumList :: [Int] -> Int
sumList xs = foldr (?op) (?base) xs

catList :: [String] -> String
catList xs = foldr (?op) (?base) xs
```

Executing `foldr`

To develop some intuition about `foldr` lets "run" it a few times by hand.

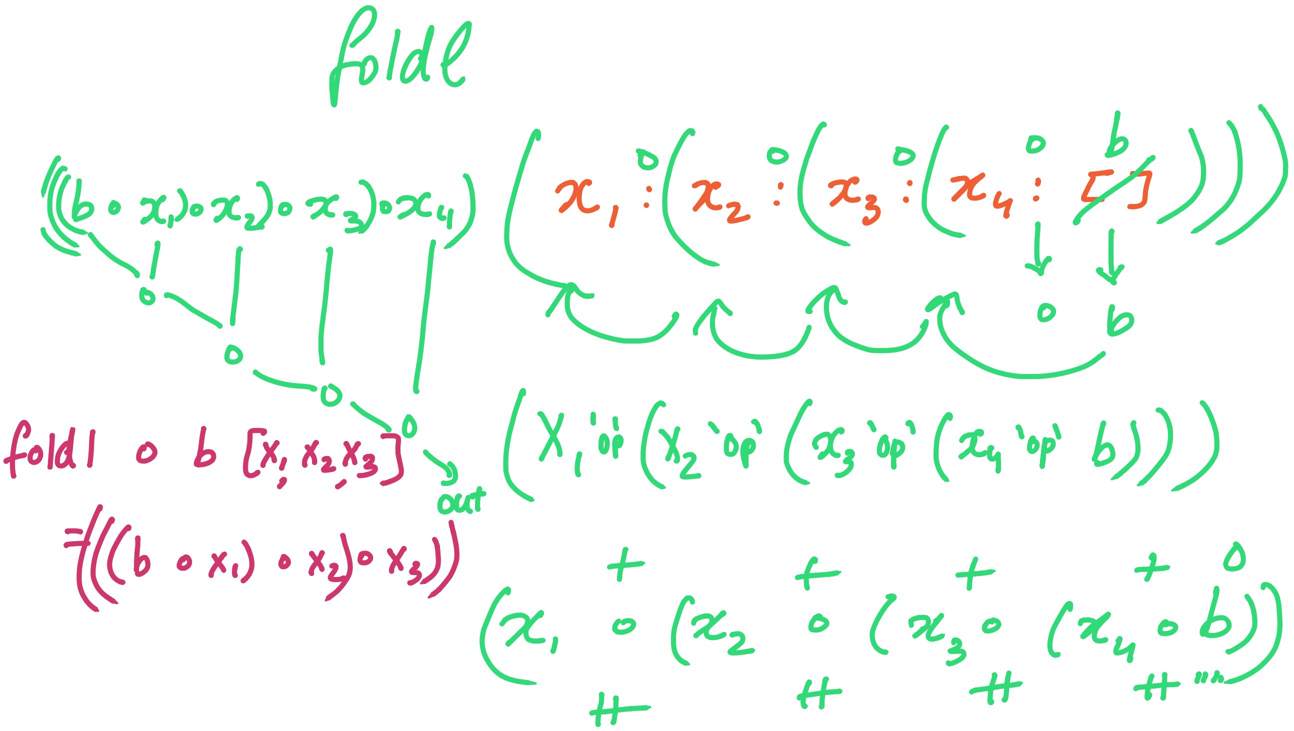
```
foldr op b (a1:a2:a3:a4:[])
==>
  a1 `op` (foldr op b (a2:a3:a4:[]))
==>
  a1 `op` (a2 `op` (foldr op b (a3:a4:[])))
==>
  a1 `op` (a2 `op` (a3 `op` (foldr op b (a4:[]))))
==>
  a1 `op` (a2 `op` (a3 `op` (a4 `op` foldr op b [])))
==>
  a1 `op` (a2 `op` (a3 `op` (a4 `op` b)))
```

Look how it *mirrors* the structure of lists!

- `(:)` is replaced by `op`
- `[]` is replaced by `base`

So

```
foldr (+) 0 (x1:x2:x3:x4:[])
==> x1 + (x2 + (x3 + (x4 + 0)))
```



Typing `foldr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op base []     = base
foldr op base (x:xs) = op x (foldr op base xs)
```

`foldr` takes as input

- a *reducer* function of type `a -> b -> b`
- a *base* value of type `b`
- a *list* of values to reduce `[a]`

and returns as output

- a *reduced* value `b`

$$(a \to b \to b) \to b \to [a] \to b$$

$$\text{foldr}\ op\ b\ (a_1 : a_2 : a_3 : a_4 : [\,])$$

$$=$$

$$(a_1 \text{`o`} (a_2 \text{`o`} (a_3 \text{`o`} \underbrace{(a_4 \text{`o`} b)})))$$

$$o :: a \to b \to b$$

QUIZ

Recall the function to compute the `len` of a list

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

Which of these is a valid implementation of ~~Len~~ len

✗ **A.** `len = foldr (\n -> n + 1) 0`

B. `len = foldr (\n m -> n + m) 0`

C. `len = foldr (\_ n -> n + 1) 0`

D. `len = foldr (\x xs -> 1 + len xs) 0`

E. All of the above

$$\underline{x_1} : x_2 : (x_3 : (\underline{x_4} : []))$$

$$\downarrow$$

$$(1 + \underbrace{(1 + (1 + (1 + 0)))}_{v})$$

$$(\backslash x\ \underline{v} \to 1 + \underline{v})$$

The Missing Parameter Revisited

We wrote `foldr` as

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op base []     = base
foldr op base (x:xs) = op x (foldr op base xs)
```

but can also write this

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op base  = go
  where
     go []     = base
     go (x:xs) = op x (go xs)
```

Can someone explain where the `xs` went *missing* ?

Trees

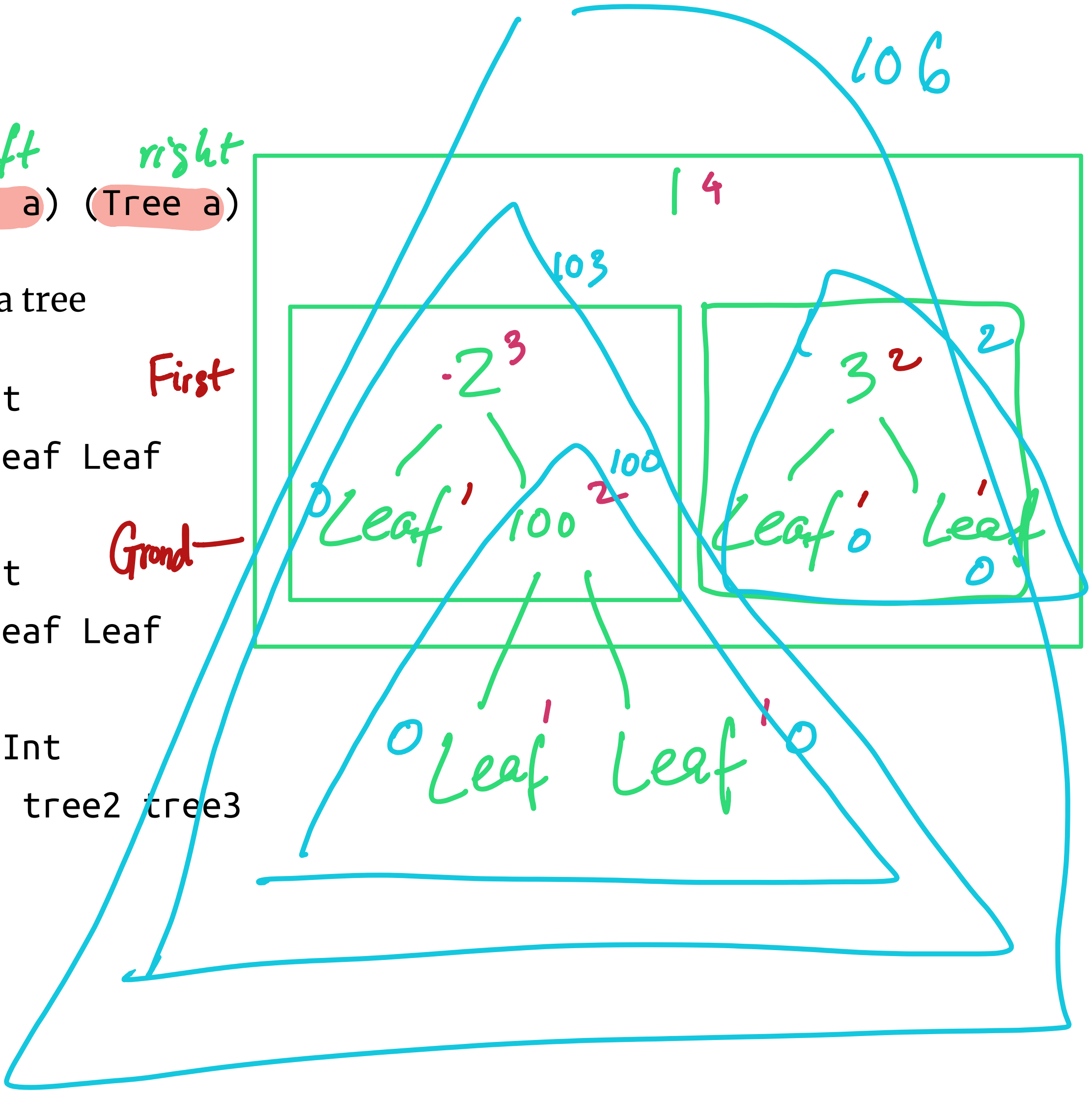
Recall the `Tree a` type from last time

left right

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

For example here's a tree

```haskell
tree2 :: Tree Int
tree2 = Node 2 Leaf Leaf

tree3 :: Tree Int
tree3 = Node 3 Leaf Leaf

tree123 :: Tree Int
tree123 = Node 1 tree2 tree3
```



Some Functions on Trees

Lets write a function to compute the `height` of a tree

```
height :: Tree a -> Int
height Leaf         = 0
height (Node x l r) = 1 + max (height l) (height l)
```

Here's another to *sum* the leaves of a tree:

```
sumTree :: Tree Int -> Int
sumTree Leaf         = ???
sumTree (Node x l r) = ???
```

Gathers all the elements that occur as leaves of the tree:

```
toList :: Tree a -> [a]
toList Leaf         = ???
toList (Node x l r) = ???
```

Lets give it a whirl

```
>>> height tree123
2

>>> sumTree tree123
6

>>> toList tree123
[1,2,3]
```

Pattern: Tree Fold

Can you spot the pattern? Those three functions are almost the same!

Step 1: Rename to maximize similarity

```
-- height
foo Leaf         = 0
foo (Node x l r) = 1 + max (foo l) (foo l)

-- sumTree
foo Leaf         = 0
foo (Node x l r) = foo l + foo r

-- toList
foo Leaf         = []
foo (Node x l r) = x : foo l ++ foo r
```

Step 2: Identify the differences

1. ???
2. ???

Step 3 Make *differences* a parameter

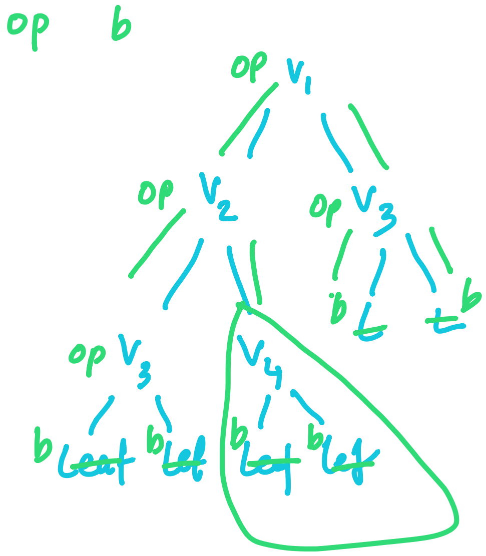
```
foo p1 p2 Leaf         = ???
foo p1 p2 (Node x l r) = ???
```

Pattern: Folding on Trees

```
tFold op b Leaf         = b
tFold op b (Node x l r) = op x (tFold op b l) (tFold op b r)
```

Lets try to work out the type of `tFold`!

```
tFold :: t_op -> t_b -> Tree a -> t_out
```



QUIZ

Suppose that `t :: Tree Int`.

What does `tFold (\x y z -> y + z) 1 t` return?

a. `0`

b. the *largest* element in the tree `t`

c. the *height* of the tree `t`

d. the *number-of-leaves* of the tree `t`

e. type *error*

EXERCISE

Write a function to compute the *largest* element in a tree or `0` if tree is empty or all negative.

```
treeMax :: Tree Int -> Int
treeMax t = tFold f b t
  where
     f    = ???
     b    = ???
```

Map over Trees

We can also write a `tmap` equivalent of `map` for `Tree` s

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf x)   = Leaf (f x)
treeMap f (Node l r) = Node (treeMap f l) (treeMap f r)
```

which gives

```
>>> treeMap (\n -> n * n) tree123     -- square all elements
of tree
Node 1 (Node 4 Leaf Leaf) (Node 9 Leaf Leaf)
```

EXERCISE

Recursion is **HARD TO READ** do we really have to use it ?

Lets rewrite `treeMap` using `tFold` !

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f t = tFold op base t
  where
     op    = ???
     base  = ???
```

When you are done, we should get

```
>>> animals = Node "cow" (Node "piglet" Leaf Leaf) (Leaf "hip
po" Leaf Leaf)
>>> treeMap reverse animals
Node "woc" (Node "telgip" Leaf Leaf) (Leaf "oppih" Leaf Leaf)
```

WED NOV 4

NEW DEADLINE for 01-TREES

Examples: `foldDir`

```
data Dir a
  = Fil a                   -- ^ A single file named `a`
  | Sub a [Dir a]           -- ^ A sub-directory name `a` with cont
ents `[Dir a]`

data DirElem a
  = SubDir a                -- ^ A single Sub-Directory named `a`
  | File a                  -- ^ A single File named `a`

foldDir :: ([a] -> r -> DirElem a -> r) -> r -> Dir a -> r
foldDir f r0 dir = go [] r0 dir
  where
      go stk r (Fil a)    = f stk r (File a)
      go stk r (Sub a ds) = L.foldl' (go stk') r' ds
        where
            r'   = f stk r (SubDir a)
            stk' = a:stk
```

`foldDir` takes as input

- an *accumulator* `f` of type `[a] -> r -> DirElem a -> r`
    - takes as *input* the path `[a]` , the current result `r` , the next `DirElem [a]`
    - and returns as *output* the new result `r`

- an *initial* value of the result `r0` and
- directory to fold over `dir`

And returns the result of running the *accumulator* over the whole `dir`.

Examples: Spotting Patterns In The "Real" World

These patterns in "toy" functions appear regularly in "real" code

1. Start with beginner's version riddled with explicit recursion (swizzle-v0.html).
2. Spot the patterns and eliminate recursion using HOFs (swizzle-v1.html).
3. Finally refactor the code to "swizzle" and "unswizzle" without duplication (swizzle-v2.html).

Try it yourself

- Rewrite the code that swizzles `Char` to use the `Map k v` type in `Data.Map`

Which is more readable? HOFs or Recursion

At first, *recursive* versions of `shout` and `squares` are easier to follow

- `fold` takes a bit of getting used to!

With practice, the *higher-order* versions become easier

- only have to understand specific operations
- recursion is lower-level & have to see "loop" structure
- worse, potential for making silly off-by-one errors

2004

Indeed, HOFs were the basis of `map/reduce` and the big-data revolution (http://en.wikipedia.org/wiki/MapReduce)

- Can *parallelize* and *distribute* computation patterns just once (https://www.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf)
- Reuse (http://en.wikipedia.org/wiki/MapReduce) across hundreds or thousands of instances!

HOFS FTW!

02 TRES DUE
NOV 4

01-trees due Friday 11/6

Haskell Crash Course Part III

Writing Applications

Lets write the classic "Hello world!" program.

For example, in Python you may write:

```
def main():
    print "hello, world!"

main()
```

and then you can run it:

```
$ python hello.py
hello world!
```

*Haskell is a **Pure** language.*

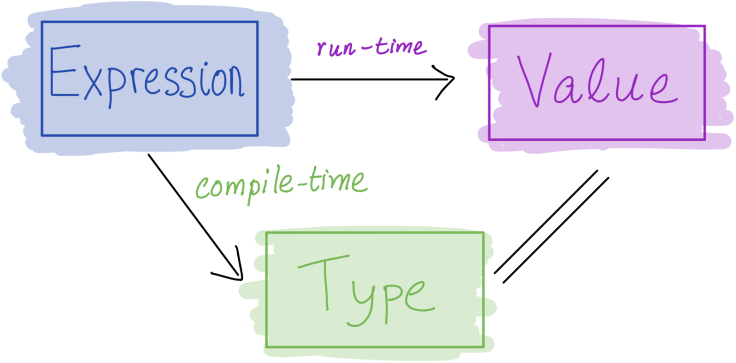
Not a *value* judgment, but a precise *technical* statement:

The "Immutability Principle":

- A function must *always* return the same output for a given input
- A function's behavior should *never change*

foo :: In → Out

No Side Effects



Haskell's most radical idea: `expression` =*> `value`

- When you evaluate an expression you get a value and
- **Nothing else happens**

"pure"

Specifically, evaluation must not have an **side effects**

anything OTHER THAN the OUTPUT VALUE

- *change* a global variable or
- *print* to screen or
- *read* a file or
- *send* an email or
- *launch* a missile.

OK in python

But… how to write "Hello, world!"

But, we **want** to …

- **print** to screen
- **read** a file
- **send** an email

Thankfully, you *can* do all the above via a very clever idea: `Recipe`

Recipes

This analogy is due to Joachim Brietner (https://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html)

Haskell has a special type called `IO` – which you can think of as `Recipe`

```
type Recipe a = IO a
```

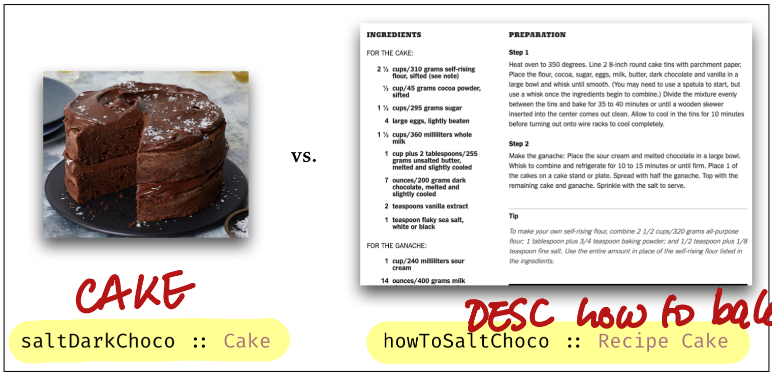
A *value* of type `Recipe a`

- is a **description** of a *computation* that can have *side-effects*
- which **when executed** performs some effectful I/O operations
- to **produce** a value of type `a`.

Recipes have No Side Effects

A value of type `Recipe a` is

- A **description** of a computation that can have side-effects



INGREDIENTS

FOR THE CAKE:

- 2 ½ cups/310 grams self-rising flour, sifted (see note)
- ½ cup/45 grams cocoa powder, sifted
- 1 ½ cups/295 grams sugar
- 4 large eggs, lightly beaten
- 1 ½ cups/360 milliliters whole milk
- 1 cup plus 2 tablespoons/255 grams unsalted butter, melted and slightly cooled
- 7 ounces/200 grams dark chocolate, melted and slightly cooled
- 2 teaspoons vanilla extract
- 1 teaspoon flaky sea salt, white or black

FOR THE GANACHE:

- 1 cup/240 milliliters sour cream
- 14 ounces/400 grams milk

PREPARATION

Step 1

Heat oven to 350 degrees. Line 2 8-inch round cake tins with parchment paper. Place the flour, cocoa, sugar, eggs, milk, butter, dark chocolate and vanilla in a large bowl and whisk until smooth. (You may need to use a spatula to start, but use a whisk once the ingredients begin to combine.) Divide the mixture evenly between the tins and bake for 35 to 40 minutes or until a wooden skewer inserted into the center comes out clean. Allow to cool in the tins for 10 minutes before turning out onto wire racks to cool completely.

Step 2

Make the ganache: Place the sour cream and melted chocolate in a large bowl. Whisk to combine and refrigerate for 10 to 15 minutes or until firm. Place 1 of the cakes on a cake stand or plate. Spread with half the ganache. Top with the remaining cake and ganache. Sprinkle with the salt to serve.

Tip

To make your own self-rising flour, combine 2 1/2 cups/320 grams all-purpose flour; 1 tablespoon plus 3/4 teaspoon baking powder; and 1/2 teaspoon plus 1/8 teaspoon fine salt. Use the entire amount in place of the self-rising flour listed in the ingredients.

CAKE

`saltDarkChoco :: Cake`

DESC how to bake a Cake

`howToSaltChoco :: Recipe Cake`

Cake vs. Recipe

(L) chocolate *cake*, **(R)** a *sequence of instructions* on how to make a cake.

They are different (*Hint*: only one of them is delicious.)

Merely having a `Recipe Cake` has no effects! The recipe

- Does not make your oven *hot*
- Does not make your your floor *dirty*

Only One Way to Execute Recipes

Haskell looks for a special value

```
main :: Recipe ()
```

The value associated with `main` is handed to the **runtime system and executed**

Baker Aker

The Haskell runtime is a *master chef* who is the only one allowed to cook!

How to write an App in Haskell

Make a `Recipe ()` that is handed off to the master chef `main`.

- `main` can be arbitrarily complicated
- composed of **smaller** sub-recipes

A Recipe to Print to Screen

```
putStrLn :: String -> Recipe ()
```

The function `putStrLn`

- takes as input a `String`
- returns as output a `Recipe ()`

`putStrLn msg` is a `Recipe ()` – *when executed* prints out `msg` on the screen.

```
main :: Recipe ()
main = putStrLn "Hello, world!"
```

... and we can compile and run it

```
$ ghc --make hello.hs
$ ./hello
Hello, world!
```

QUIZ: How to Print Multiple Things?

Suppose I want to print *two* things e.g.

```
$ ghc --make hello.hs
$ ./hello2
Hello!
World!
```

Can we try to compile and run this:

```
main = (putStrLn "Hello!", putStrLn "World!")
```

Recipe()   Recipe()

A. Yes!

[✓ **B.**] No, there is a type error!

C. No, it compiles but produces a different result!

A Collection of Recipes

Is just ... a *collection* of Recipes!

```
recPair :: (Recipe (), Recipe ())
recPair = (putStrLn "Hello!", putStrLn "World!")

recList :: [Recipe ()]
recList = [putStrLn "Hello!", putStrLn "World!"]
```

... we need a way to **combine** recipes!

Combining? Just `do` *it!*

We can *combine* many recipes into a single one using a `do` block

```
foo :: Recipe a3
foo = do r1         -- r1 :: Recipe a1
         r2         -- r2 :: Recipe a2
         r3         -- r3 :: Recipe a3
```

(or if you *prefer* curly braces to indentation)

```
foo = do { r1;     -- r1 :: Recipe a1
           r2;     -- r2 :: Recipe a2
           r3      -- r3 :: Recipe a3
         }
```

The **do** block combines sub-recipes `r1`, `r2` and `r3` into a *new* recipe that

- Will execute each sub-recipe in *sequence* and
- Return the value of type `a3` produced by the last recipe `r3`

Combining? Just **do** *it!*

So we can write

```
main = do putStrLn "Hello!"
          putStrLn "World!"
```

or if you prefer

```
main = do { putStrLn "Hello!";
            putStrLn "World!"
          }
```

EXERCISE: Combining Many Recipes

Write a function called `sequence` that

- Takes a non-empty *list* of recipes `[r1,...,rn]` as input and
- Returns a *single* recipe equivalent to `do {r1; ...; rn}`

```
sequence :: [Recipe a] -> Recipe a
sequence rs = ???
```

When you are done you should see the following behavior

```
-- Hello.hs

main = sequence [putStrLn "Hello!", putStrLn "World!"]
```

and then

```
$ ghc --make Hello.hs
$ ./hello
Hello!
World!
```

Using the Results of (Sub-) Recipes

Suppose we want a function that **asks** for the user's name

Print / putStrLn

getline

```
$ ./hello
What is your name?
Ranjit                # <<<<< user enters
Hello Ranjit!
```

PutStrln

We can use the following sub-recipes

```
-- | read and return a line from stdin as String
getLine  :: Recipe String

-- take a string s, return a recipe that prints  s
putStrLn :: String -> Recipe ()
```

But how to

- *Combine* the two sub-recipes while
- *Passing* the result of the first sub-recipe to the second.

'getline' 'putStrln'

Naming Recipe Results via "Assignment"

You can write

```
x <- recipe
```

to *name* the result of executing `recipe`

- `x` can be used to refer to the result in *later* code

Naming Recipe Results via "Assignment"

Lets, write a function that *asks* for the user's name

```
main = ask

ask :: Recipe ()
ask = do name <- getLine;
         putStrLn ("Hello " ++ name ++ "!")
```

Which produces the desired result

```
$ ./hello
What is your name?
Ranjit              # user enters
Hello Ranjit!
```

EXERCISE

Modify the above code so that the program *repeatedly* asks for the users's name *until* they provide a *non-empty* string.

```
-- Hello.hs

main = repeatAsk

repeatAsk :: Recipe ()
repeatAsk = _fill_this_in


isEmpty :: String -> Bool
isEmpty s = length s == 0
```

When you are done you should get the following behavior

```
$ ghc --make hello.hs

$ ./hello
What is your name?
# user hits return
What is your name?
# user hits return
What is your name?
# user hits return
What is your name?
Ranjit  # user enters
Hello Ranjit!
```

Non-empty String

EXERCISE

Modify your code to *also* print out a **count** in the prompt

```
$ ghc --make hello.hs

$ ./hello
(0) What is your name?
                          # user hits return
(1) What is your name?
                          # user hits return
(2) What is your name?
                          # user hits return
(3) What is your name?
Ranjit                    # user enters
Hello Ranjit!
```

That's all about IO

You should be able to implement `build` from `Directory.hs`

Using these library functions imported at the top of the file

```
import System.FilePath   (takeDirectory, takeFileName, (</>))
import System.Directory  (doesFileExist, listDirectory)
```

The functions are

- `takeDirectory`
- `takeFileName`
- `(</>)`
- `doesFileExist`
- `listDirectory`

`hoogle` the documentation to learn about how to use them.

(https://ucsd-cse230.github.io/fa20/feed.xml)
(https://twitter.com/ranjitjhala)
(https://plus.google.com/u/0/104385825850161331469)
(https://github.com/ranjitjhala)

Generated by Hakyll (http://jaspervdj.be/hakyll), template by Armin Ronacher (http://lucumr.pocoo.org), suggest improvements here (https://github.com/ucsd-progsys/liquidhaskell-blog/).