# *Bottling Computation Patterns*

## *Polymorphism and Equational Abstractions 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*

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.

# *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 :: [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))))
```

(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)))
```

map op xs = case xs of

Nil → Nil ✓

(Cons h t) → Cons $\underbrace{(op\ h)}_{T_o}$ (map op t)

$T_h$

map :: ? $T_{op} \to T_{xs} \to T_{body}$

t :: List $T_h$

$T_{xs}$ = List $T_h$

$T_{body}$ = List $T_o$

$T_{op}$ = $T_h \to T_o$

map :: $(T_h \to T_o) \to$ List $T_h \to$ List $T_o$

map :: $(a \to b) \to$ List $a \to$ List $b$

# 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`

# *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`

**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

# *The Missing Parameter Revisited*

We wrote `foldr` as

```
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

```
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

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

For example here's a tree

```
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 "hippo" Leaf Leaf)
>>> treeMap reverse animals
Node "woc" (Node "telgip" Leaf Leaf) (Leaf "oppih" Leaf Leaf)
```

map/reduce "google" OSDI 2004

Higher Order func

## *Examples:* `foldDir`

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

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

directory

```
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
```

path-to-elem cur nextdirelem

`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

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!

---