Pattern: Reduction

Computation patterns are everywhere let's revisit our old `sumList`

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

Next, a function that concatenates the Strings in a list

```
catList :: [String] -> String
catList [] = ""
catList (x:xs) = x ++ (catList xs)
```
Let's 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. ??? 0 vs ""
2. ??? + vs #-

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 base (x1:x2:x3:x4:[])

==> 
ex1 `op` (foldr op base (x2:x3:x4:[]))

==> 
ex1 `op` (x2 `op` (foldr op base (x3:x4:[])))

==> 
ex1 `op` (x2 `op` (x3 `op` (foldr op base (x4:[])))))

==> 
ex1 `op` (x2 `op` (x3 `op` (x4 `op` foldr op base [])))

==> 
ex1 `op` (x2 `op` (x3 `op` (x4 `op` base)))

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 \textit{foldr}

\begin{verbatim}
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op base []    = base
foldr op base (x:xs) = op x (foldr op base xs)
\end{verbatim}

\textit{foldr} takes as input

- a \textit{reducer} function of type \( a \rightarrow b \rightarrow b \)
- a \textit{base} value of type \( b \)
- a \textit{list} of values to reduce \([a]\)

and returns as output

- a \textit{reduced} value \( b \)
QUIZ

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

\[
\text{len} \ [\ ] = 0
\]
\[
\text{len} \ (x:xs) = 1 + \text{len} \ xs
\]

Which of these is a valid implementation of `listLen`

A. `len = foldr (\n -> n + 1) 0`  \( \times \) \text{`f` takes 1 arg!}

B. `len = foldr (+ 1) 0`  \( \times \) \text{computes sum!}

C. `len = foldr (\_ n -> n + 1) 0`  \( \checkmark \)
The Missing Parameter Revisited

We wrote foldr as

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]
\[
\text{foldr \_ op base \_} = \text{base}
\]
\[
\text{foldr \_ op base (x:xs)} = \text{op x (foldr \_ op base xs)}
\]

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

\[
\text{height :: Tree } a \rightarrow \text{ Int } \\
\text{height Leaf } = 0 \\
\text{height (Node } x \text{ l } r) = 1 + \max (\text{height l}, \text{height r})
\]

Here’s another to sum the leaves of a tree:

\[
\text{sumTree :: Tree } \text{ Int } \rightarrow \text{ Int } \\
\text{sumTree Leaf } = ??? \\
\text{sumTree (Node } x \text{ l } r) = ???
\]

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

\[
\text{toList :: Tree } a \rightarrow [a] \\
\text{toList Leaf } = [] \\
\text{toList (Node } x \text{ l } r) = ???
\]

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

\[
\text{tFold op b Leaf} = b \times \\
\text{tFold op b (Node x l r)} = \text{op x (tFold op b l) (tFold op b r)}
\]

Lets try to work out the type of tFold!

\[
\text{tFold :: t_op} \to \text{t_b} \to \text{Tree a} \to \text{t_out}
\]
**QUIZ**

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

a. 0

b. the *largest* element in the tree  

c. the *height* of the tree  

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

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

```haskell
>>> 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

\[
\text{treeMap } f \ t = \text{tFold } \text{op base } t
\]

\[
\text{where}
\]

\[
\text{op } = \text{???}
\]

\[
\text{base } = \text{???}
\]

When you are done, we should get

\[
\text{>>> animals } = \text{Node } "\text{cow}" (\text{Node } "\text{piglet}" \text{ Leaf Leaf}) (\text{Leaf } "\text{hippo}" \text{ Leaf Leaf})
\]

\[
\text{>>> treeMap reverse animals}
\]

\[
\text{Node } "\text{woc}" (\text{Node } "\text{telgip}" \text{ Leaf Leaf}) (\text{Leaf } "\text{oppih}" \text{ Leaf Leaf})
\]

**Examples: Spotting Patterns In The “Real” World**

We saw patterns in “toy” functions.
But these patterns appear regularly in “real” code - look for them!

For an example, see the below

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!
