Bottling Computation Patterns

Polymorphism and HOFs are the Secret Sauce

Refactor arbitrary repeated code patterns ...
... into precisely specified and reusable functions

\[
\text{List } a = \text{Nil} \\
1 \text{ Cons } a \ (\text{List } a)
\]

\[
\text{doTwice } f \ x = f \ (f \ x)
\]

EXERCISE: Iteration

Write a function that squares a list of Int
squares :: [Int] -> [Int]
squares ns = ???

When you are done you should see

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

**Pattern: Iteration**

Next, let's write a function that converts a String to uppercase.

```haskell
>>> shout "hello"
"HELLO"
```
Recall that in Haskell, a `String` is just a `[Char]`.

```haskell
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*

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

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

**Step 2** Identify what is *different*
• In squares we transform \( x \) to \( x \times x \)

• In shout we transform \( x \) to `Data.Char.toUpper x`

**Step 3** Make **differences** a parameter

• Make `transform` a parameter \( f \)

\[
\text{foo } f \ [ ] \ = \ [ ] \\
\text{foo } f \ (x:xs) \ = \ (f \ x) : \text{foo } f \ xs
\]

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

\[
\text{map } f \ [ ] \ = \ [ ] \\
\text{map } f \ (x:xs) \ = \ (f \ x) : \text{map } f \ xs
\]

\( \text{map} \) bottles the common pattern of iteratively transforming a list:

Fairy In a Bottle
QUIZ

What is the type of \texttt{map}?

\begin{verbatim}
map :: ???
map f [] = []
map f (x:xs) = (f x) : map f xs
\end{verbatim}

\begin{itemize}
  \item A. (Int -> Int) -> [Int] -> [Int]
  \item B. (a -> a) -> [a] -> [a]
  \item C. [a] -> [b]
  \item D. (a -> b) -> [a] -> [b]
  \item E. (a -> b) -> [a] -> [a]
\end{itemize}
The type precisely describes \texttt{map}

\begin{verbatim}
>>> :type map
map :: (a -> b) -> [a] -> [b]
\end{verbatim}

That is, \texttt{map} takes two inputs

- a \textit{transformer} of type \texttt{a -> b}
- a \textit{list} of values \texttt{[a]}

and it returns as output

- a list of values \texttt{[b]}

that can only come by applying \texttt{f} to each element of the input list.
Reusing the Pattern

Lets reuse the pattern by *instantiating* the transformer

\textbf{shout}

\textit{-- OLD with recursion}

\texttt{shout :: [Char] \to [Char]}
\texttt{shout [] = []}
\texttt{shout (x:xs) = Char.toUpper x : shout xs}

\textit{-- NEW with \texttt{map}}

\texttt{shout :: [Char] \to [Char]}
\texttt{shout xs = map (???) xs}

\textit{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

```haskell
shout :: [Char] -> [Char]
shout = map Char.toUpperCase
```

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`

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

Next, a function that concatenates the Strings in a list

```haskell
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. `0` vs `""
2. `+` vs `#`

**Step 3 Make differences a parameter**

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