CSE 116: Fall 2019 Introduction to Functional	
Programming	
Higher-Order Functions	
Owen Arden	
UC Santa Cruz	
Based on course materials developed by Nadia Polikarpova	

Plan for this week

Last week:

- user-defined data types
 and how to manipulate them using pattern matching and recursion
- how to make recursive functions more efficient with *tail* recursion

2

3

This week:

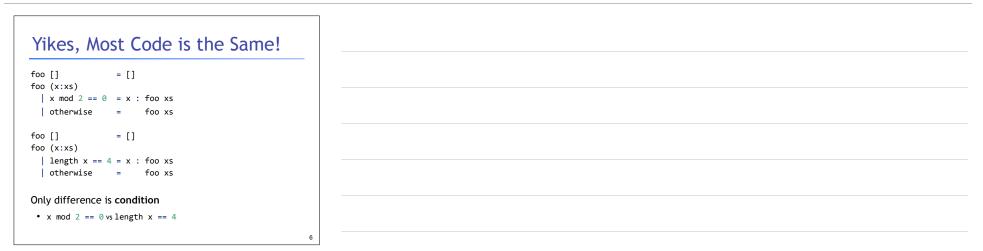
- code reuse with higher-order functions (HOFs)
- some useful HOFs: map, filter, and fold

Recursion is good

- Recursive code mirrors recursive data
 - Base constructor -> Base case
 - Inductive constructor -> Inductive case (with recursive call)
- But it can get kinda repetitive!

Example: evens
Let's write a function evens:
evens [] ==> [] evens [1,2,3,4] ==> [2,4] evens :: [Int] -> [Int] evens [] = evens (x:xs) =

Example: four-letter words	
<pre>Let's write a function fourChars: fourChars [] ==> [] fourChars ["i", "must", "do", "work"] ==> ["must", "work"]</pre>	
<pre>fourChars :: [String] -> [String] fourChars [] = fourChars (x:xs) =</pre>	
5	

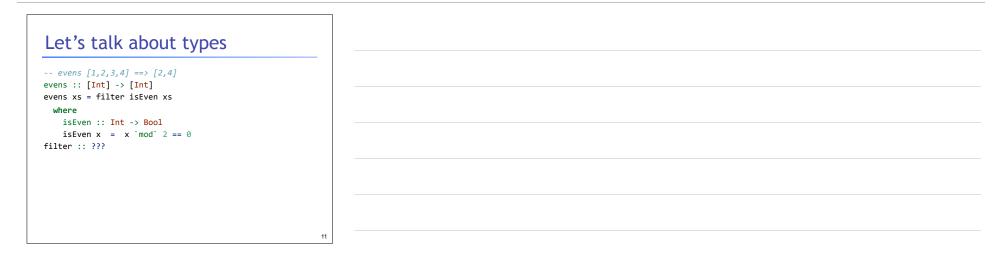


Moral of the day	
D.R.Y. Don't Repeat Yourself!	
Can we	
 reuse the general pattern and substitute in the custom condition? 	
7	

HOFs to the rescue!	
General Pattern	
 expressed as a higher-order function takes customizable operations as arguments 	
Specific Operation	
• passed in as an argument to the HOF	
	8

The "filter" pattern	
evens [] = [] evens (x:xs) [] x `mod` 2 == 0 = x : evens xs otherwise = evens xs fourChars [] = [] fourChars (x:xs) l ength x == 4 = x : fourChars xs otherwise =	
<pre>filter f [] = [] filter f (x:xs) f x = x : filter f xs otherwise = filter f xs</pre>	
Use the filter pattern to avoid duplicating code!	

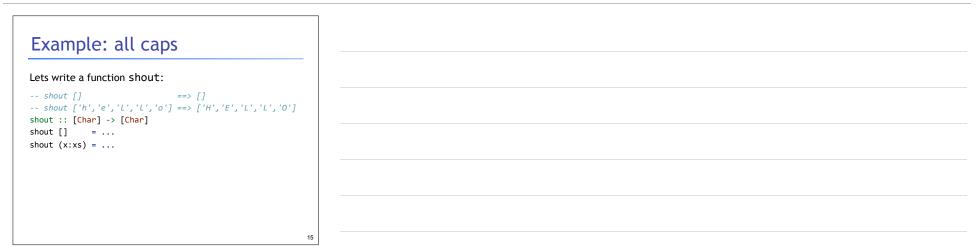
The "filter" pattern	
General Pattern	
 HOF filter Recursively traverse list and pick out elements that satisfy a predicate 	
Specific Operation	
 Predicates isEven and isFour 	
<pre>filter f [] = [] filter f (x:xs)</pre>	
f x = x : filter f xs otherwise = filter f xs	
evens = filter isEven where isEven x = x `mod` 2 == 0 isFour x = length x == 4	
10	

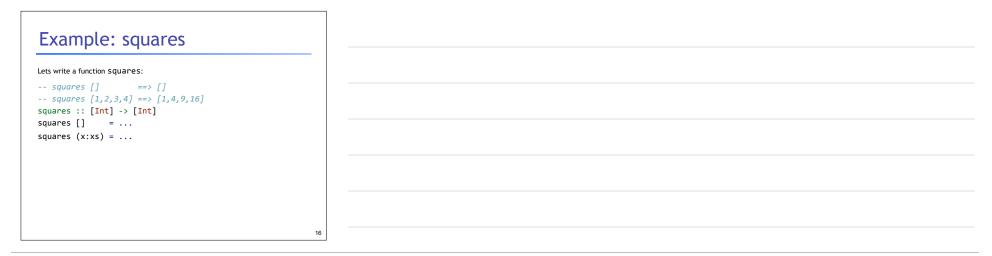


Let's talk about types				
evens [1,2,3,4] ==> [2,4] evens :: [Int] -> [Int] evens xs = filter isEven xs				
<pre>where isEven :: Int -> Bool isEven x = x `mod` 2 == 0 filter :: ???</pre>				
	12			

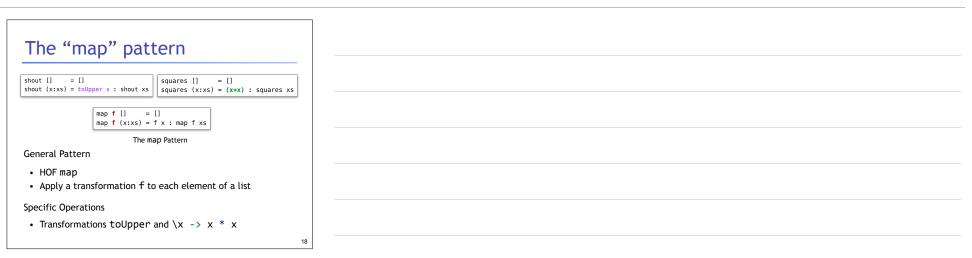
Let's talk about types	
<pre> fourChars ["i", "must", "do", "work"] ==> ["must", "work"] fourChars :: [String] -> [String]</pre>	
<pre>fourChars xs = filter isFour xs where isFour :: String -> Bool</pre>	
<pre>isFour x = length x == 4 filter :: ???</pre>	
13	

Let's talk about types		
Uh oh! So what's the type of filter?		
<pre>filter :: (Int -> Bool) -> [Int] -> [Int] ???</pre>		
<pre>filter :: (String -> Bool) -> [String] -> [String] ???</pre>		
 It does not care what the list elements are as long as the predicate can handle them 		
• It's type is polymorphic (generic) in the type of list elements		
For any type `a` if you give me a predicate on `a`s and a list of `a`s,		
I'll give you back a list of `a`s		
filter :: (a -> Bool) -> [a] -> [a]		
14		

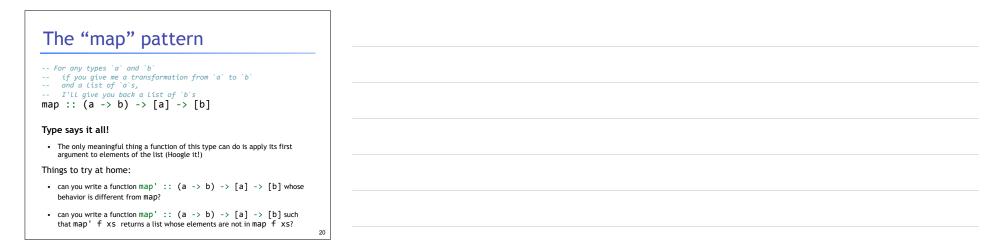




Yikes, Most Code is the Same!	
Lets rename the functions to foo:	
shout foo [] = []	
foo (x:xs) = toUpper x : foo xs	
squares	
foo [] = [] foo (x:xs) = (x * x) : foo xs	
Lets refactor into the common pattern	
pattern =	
17	



The "map" pattern	
<pre>map f [] = [] map f (x:xs) = f x : map f xs</pre>	
Lets refactor shout and squares shout = map	
squares = map	
map f [] = []	
map f(x:xs) = f x : map f xs	
shout = map ($x \rightarrow toUpper x$) squares = map ($x \rightarrow x*x$)	
19	

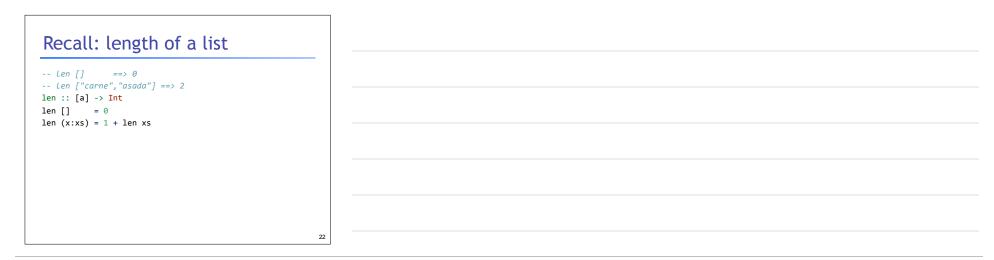


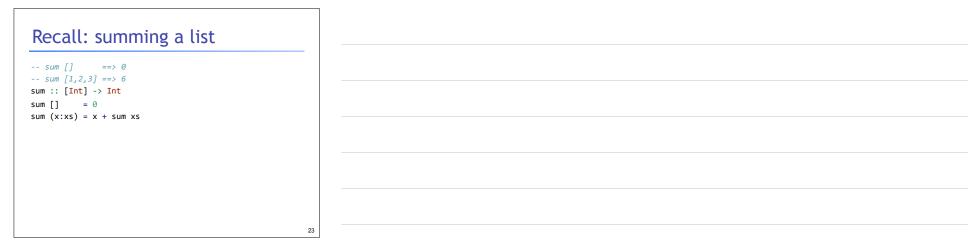
Don't Repeat Yourself

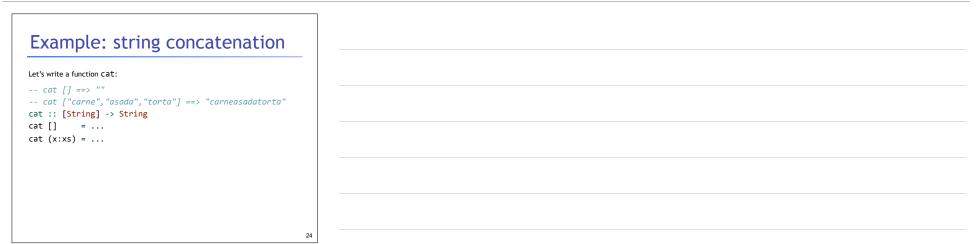
Benefits of **factoring** code with HOFs:

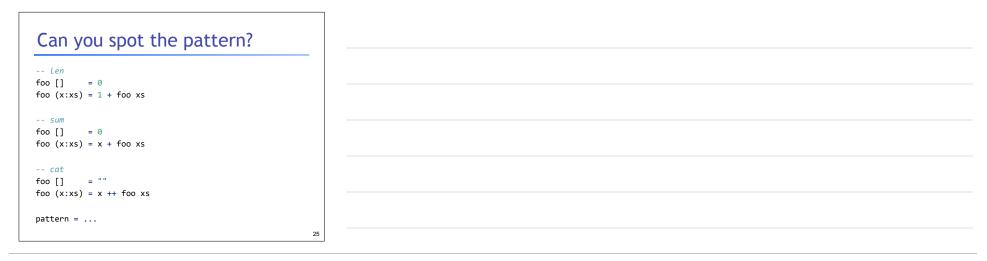
- Reuse iteration pattern
 - think in terms of standard patterns
 - less to write
 - easier to communicate
- Avoid bugs due to repetition

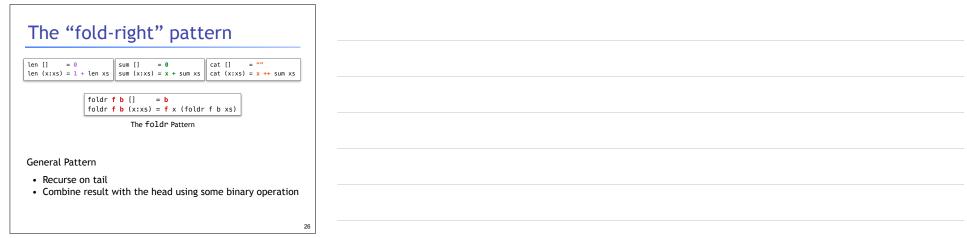
21

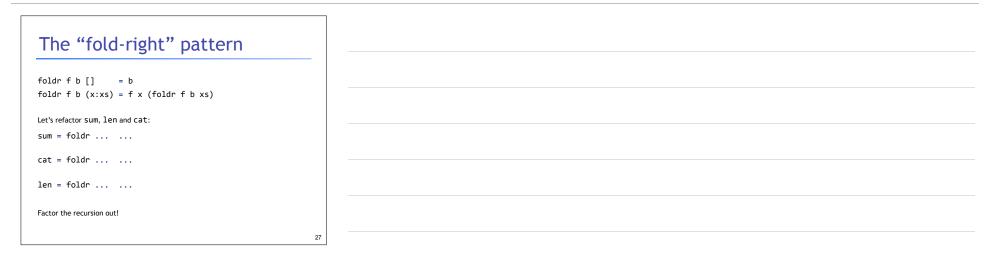


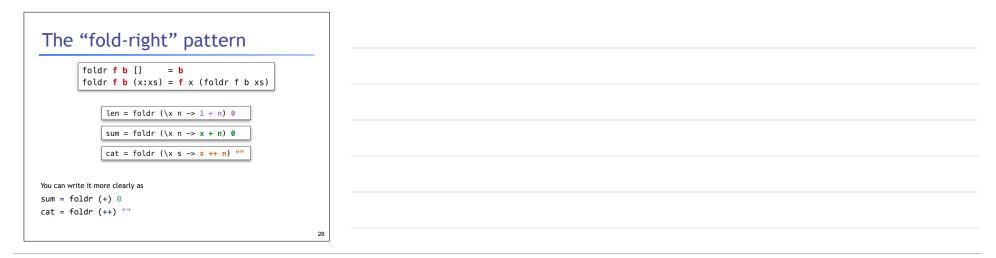


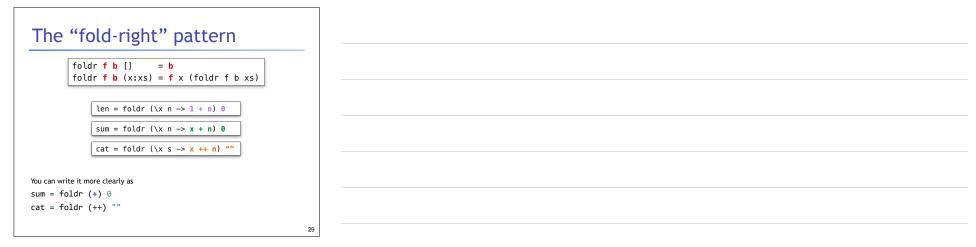


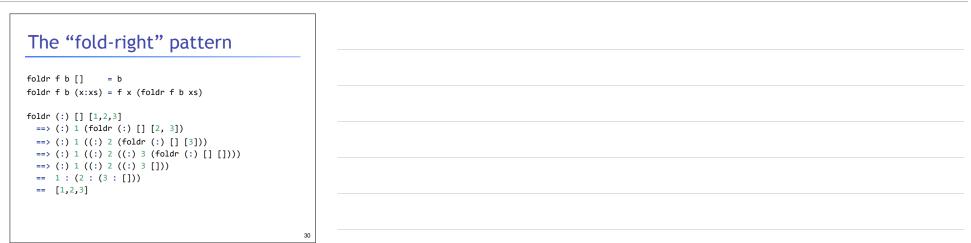


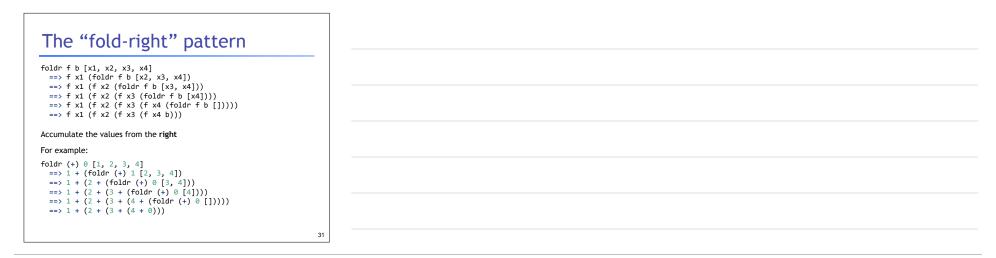








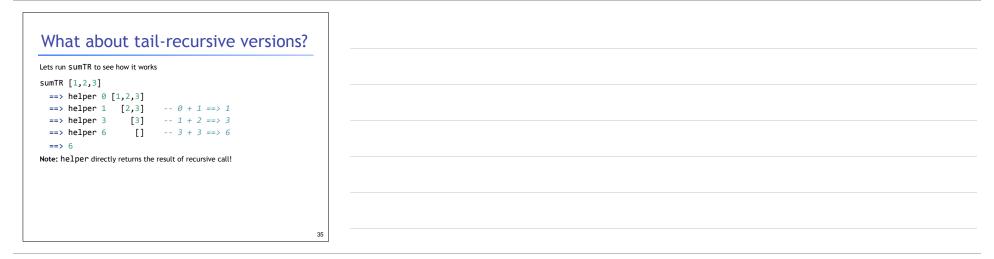




The "fold-right" pattern	
Answer: No! It calls the binary operations on the results of the recursive call	
32	

What about tail-recursive versions? Let's write tail-recursive sum! sumTR :: [Int] -> Int	
sumTR =	
33	

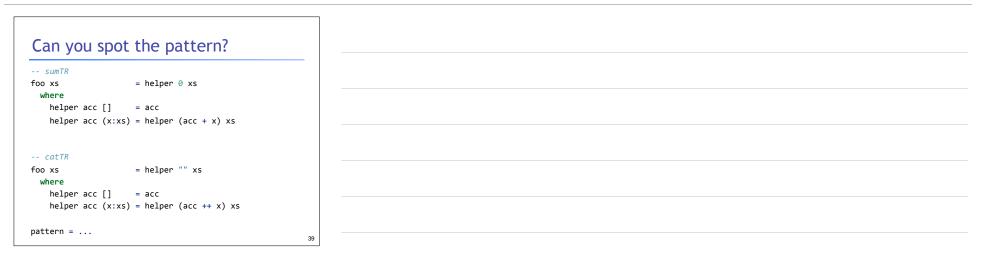
What about tail-recursive versions?	
Let's write tail-recursive sum!	
<pre>sumTR :: [Int] -> Int sumTR xs = helper 0 xs where belown pers [] = pers</pre>	
helper acc [] = acc helper acc (x:xs) = helper (acc + x) xs	
34	



What about tail-recursive versions?	
catTR :: [String] -> String catTR =	
36	

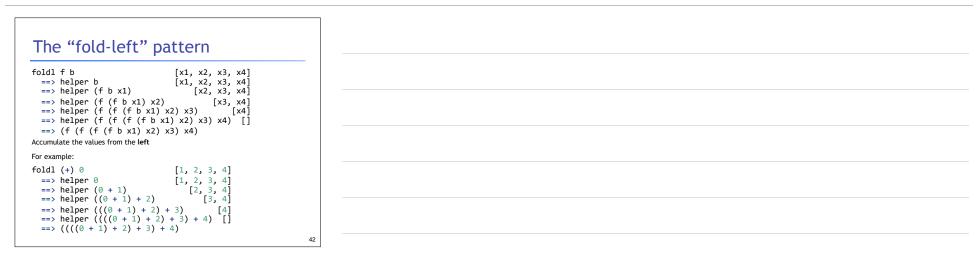
What about tail-recursive versions?	
Let's write tail-recursive cat!	
<pre>catTR :: [String] -> String catTR xs helper "" xs where</pre>	
helper acc [] = acc helper acc (x:xs) = helper (acc ++ x) xs	
37	

Lets run catTR to see how it works catTR ["carne", "asada", "torta"] ==> helper "carne" ["asada", "torta"] ==> helper "carneasada" ["torta"] ==> helper "carneasadatorta" [] ==> "carneasadatorta" [] Note: helper directly returns the result of recursive call!	<pre>catTR ["carne", "asada", "torta"] => helper "carne", "asada", "torta"] => helper "carneasada", "torta"] => helper "carneasada" ["torta"] => helper "carneasadatorta" [] => 'carneasadatorta"</pre>	What about tail-recursive versions?	
<pre>==> helper "" ["carne", "asada", "torta"] ==> helper "carneasada" ["torta"] ==> helper "carneasada" ["torta"] ==> helper "carneasadatorta" [] ==> "carneasadatorta"</pre>	<pre>==> helper "" ["carne", "asada", "torta"] ==> helper "carneasada" ["torta"] ==> helper "carneasadatorta" [] ==> "carneasadatorta"</pre>	Lets run catTR to see how it works	
Note: helper directly returns the result of recursive call!	Note: helper directly returns the result of recursive call!	<pre>==> helper "" ["carne", "asada", "torta"] ==> helper "carne" ["asada", "torta"] ==> helper "carneasada" ["torta"] ==> helper "carneasadatorta" [] ==> "carneasadatorta"</pre>	
		Note: helper directly returns the result of recursive call!	



The "fold-left" pattern	
sum xs = helper 0 xs cat xs = helper "" xs where helper acc [] = acc helper acc []	
helper acc (x:xs) = helper (acc + x) xs foldl f b xs = helper b xs where	
<pre>www.ete acc [] = acc helper acc (x:xs) = helper (f acc x) xs The foldl Pattern</pre>	
General Pattern	
• Use a helper function with an extra accumulator argument	
 To compute new accumulator, combine current accumulator with the head using some binary operation 	
40	

The "fold-left" pattern	
foldl f b xs = helper b xs where	
helper acc [] = acc helper acc (x:xs) = helper (f acc x) xs	
Let's refactor sumTR and catTR:	
sumTR = foldl	
catTR = foldl	
Factor the tail-recursion out!	
	41



Left vs. Right	
foldl f b [x1, x2, x3] ==> f (f (f b x1) x2) x3 <i>Left</i>	
foldr f b [x1, x2, x3] ==> f x1 (f x2 (f x3 b)) <i>Right</i>	
For example:	
foldl (+) 0 [1, 2, 3] ==> ((0 + 1) + 2) + 3 Left	
foldr (+) 0 [1, 2, 3] ==> 1 + (2 + (3 + 0)) Right	
Different types!	
foldl :: (b -> a -> b) -> b -> [a] -> b <i>Left</i>	
foldr :: (a -> b -> b) -> b -> [a] -> b <i>Right</i>	
43	

Useful HOF: flip	
you can write foldl (\xs x -> x : xs) [] [1,2,3]	
more concisely like so: foldl (flip (:)) [] [1,2,3]	
What is the type of flip?	
flip :: (a -> b -> c) -> b -> a -> c	
	44

Useful HOF: compose	
you can write map (\x -> f (g x)) ys	
more concisely like so:	
<pre>map (f . g) ys What is the type of (.)?</pre>	
(.) :: (b -> c) -> (a -> b) -> a -> c	
	45

Higher Order Functions	
Iteration patterns over collections:	-
 Filter values in a collection given a predicate Map (iterate) a given transformation over a collection Fold (reduce) a collection into a value, given a binary expection to experime a source. 	
operation to combine results Useful helper HOFs:	
 Flip the order of function's (first two) arguments Compose two functions 	
	46

Higher Order Functions	
HOFs can be put into libraries to enable modularityData structure library implements map, filter, fold for its	
collections	
 generic efficient implementation 	
 generic optimizations: map f (map g xs)> map (f.g) xs 	
• Data structure clients use HOFs with specific operations	
\circ $$ no need to know the implementation of the collection	
Enabled the "big data" revolution e.g. MapReduce, Spark	
	47

That's all folks!]
48	