CSE 114A

Introduction to Functional Programming

Higher-Order Functions

Based on course materials developed by Nadia Polikarpova and Owen Arden

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

This week:

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

2

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) = ...
```

4

Example: four-letter words

Let's write a function fourChars:

```
-- fourChars [] ==> []
-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars [] = ...
fourChars (x:xs) = ...
```

5

Yikes, Most Code is the Same!

Only difference is **condition**

```
• x mod 2 == 0 vs length x == 4
```

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

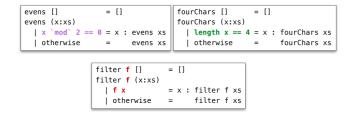
- expressed as a higher-order function
- takes customizable operations as arguments

Specific Operation

• passed in as an argument to the HOF

8

The "filter" pattern



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 is Even and is Four

```
evens = filter isEven
where
isEven x = x `mod` 2 == 0
```

```
fourChars = filter isFour
where
isFour x = length x == 4
```

.

Let's talk about types

```
-- evens [1,2,3,4] ==> [2,4]

evens :: [Int] -> [Int]

evens xs = filter isEven xs

where

   isEven :: Int -> Bool

   isEven x = x `mod` 2 == 0

filter :: ???
```

11

Let's talk about types

```
-- evens [1,2,3,4] ==> [2,4]

evens :: [Int] -> [Int]

evens xs = filter isEven xs

where

  isEven :: Int -> Bool

  isEven x = x `mod` 2 == 0

filter :: ???
```

Let's talk about types

```
-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars xs = filter isFour xs
  where
    isFour :: String -> Bool
    isFour x = length x == 4
filter :: ???
```

13

Let's talk about types

Uh oh! So what's the type of filter?

```
filter :: (Int -> Bool) -> [Int] -> [Int] -- ???

filter :: (String -> Bool) -> [String] -> [String] -- ???
```

- It does not care what the list elements are
 - $^{\circ}$ $\,$ 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

Example: all caps

Lets write a function shout:

```
-- shout [] ==> []
-- shout ['h', 'e', 'L', 'L', 'o'] ==> ['H', 'E', 'L', 'L', '0']
shout :: [Char] -> [Char]
shout [] = ...
shout (x:xs) = ...
```

Example: squares

Lets write a function squares:

```
-- squares [] ==> []

-- squares [1,2,3,4] ==> [1,4,9,16]

squares :: [Int] -> [Int]

squares [] = ...

squares (x:xs) = ...
```

16

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

The map Pattern

General Pattern

- HOF map
- Apply a transformation f to each element of a list

Specific Operations

• Transformations to Upper and $\x -> x * x$

The "map" pattern

```
map f []
            = []
map f (x:xs) = f x : map f xs
Lets refactor shout and squares
shout = map ...
squares = map ...
             map f []
                        = []
             map f(x:xs) = f x : map f xs
shout = map (\x -> toUpper x)
                           squares = map (\x -> x*x)
```

19

QUIZ

What is the type of map? *

```
\mathsf{map} \ \mathsf{f} \ [] \qquad = \ []
map f (x:xs) = f x : map f xs
(A) (Char -> Char) -> [Char] -> [Char]
(B) (Int -> Int) -> [Int] -> [Int]
(C) (a -> a) -> [a] -> [a]
```

(D) (a -> b) -> [a] -> [b]

(E) (a -> b) -> [c] -> [d]



http://tiny.cc/cse116-map-ind

QUIZ

What is the type of map? *

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





http://tiny.cc/cse116-map-grp

The "map" pattern

```
-- For any types `a` and `b`
-- if you give me a transformation from `a` to `b`
-- and a list of `a`s,
-- I'll give you back a list of `b`s
map :: (a -> b) -> [a] -> [b]
```

Type says it all!

The only meaningful thing a function of this type can do is apply its first argument to elements of the list (Hoogle it!)

Things to try at home:

- can you write a function map' :: (a -> b) -> [a] -> [b] whose behavior is different from map?
- can you write a function map' :: (a -> b) -> [a] -> [b] such that map' f xs returns a list whose elements are not in map f xs?

20

QUIZ

What is the value of quiz? *

```
map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]

quiz = map (\((x, y) \rightarrow x + y) [1, 2, 3]
```

- (A) [2, 4, 6]
- (B) [3, 5]
- (C) Syntax Error
- O (D) Type Error
- (E) None of the above



http://tiny.cc/cse116-quiz-ind

23

QUIZ

What is the value of quiz? *

```
map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]

quiz = map (\(x, y) \leftarrow x + y) [1, 2, 3]
```

- (A) [2, 4, 6]
- (B) [3, 5]
- (C) Syntax Error
- O (D) Type Error
- (E) None of the above



http://tiny.cc/cse116-quiz-grp

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

25

Recall: length of a list

```
-- len [] ==> 0

-- len ["carne", "asada"] ==> 2

len :: [a] -> Int

len [] = 0

len (x:xs) = 1 + len xs
```

26

Recall: summing a list

```
-- sum [] ==> 0

-- sum [1,2,3] ==> 6

sum :: [Int] -> Int

sum [] = 0

sum (x:xs) = x + sum xs
```

Example: string concatenation

```
Let's write a function cat:

-- cat [] ==> ""

-- cat ["carne", "asada", "torta"] ==> "carneasadatorta"

cat :: [String] -> String

cat [] = ...

cat (x:xs) = ...
```

28

Can you spot the pattern?

```
-- Len
foo [] = 0
foo (x:xs) = 1 + foo xs

-- sum
foo [] = 0
foo (x:xs) = x + foo xs

-- cat
foo [] = ""
foo (x:xs) = x ++ foo xs

pattern = ...
```

29

The "fold-right" pattern

General Pattern

- Recurse on tail
- Combine result with the head using some binary operation

The "fold-right" pattern

```
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

Let's refactor sum, len and cat:
sum = foldr ...

cat = foldr ...

len = foldr ...

Factor the recursion out!
```

31

The "fold-right" pattern

```
foldr f b [] = b
foldr f b (x:xs) = f x (foldr <math>f b xs)

len = foldr (\x n \rightarrow 1 + n) 0

sum = foldr (\x n \rightarrow x + n) 0

cat = foldr (\x x \rightarrow x + n) 0
```

32

The "fold-right" pattern

```
foldr f b [] = b

foldr f b (x:xs) = f x (foldr <math>f b xs)

len = foldr (\x n -> 1 + n) 0
sum = foldr (\x n -> x + n) 0
cat = foldr (\x s -> x + n) ""
```

You can write it more clearly as

You can write it more clearly as sum = foldr (+) 0 cat = foldr (++) ""

```
sum = foldr (+) 0
cat = foldr (++) ""
```

QUIZ

```
What does this evaluate to? *
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)
quiz = foldr (:) [] [1,2,3]
(A) Type error
(B) [1,2,3]
(C) [3,2,1]
O (D) [[3],[2],[1]]
(E) [[1],[2],[3]]
                                          http://tiny.cc/cse116-foldeval-ind
```

QUIZ

```
What does this evaluate to? *
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)
quiz = foldr (:) [] [1,2,3]
(A) Type error
(B) [1,2,3]
(C) [3,2,1]
O (D) [[3],[2],[1]]
(E) [[1],[2],[3]]
```

http://tiny.cc/cse116-foldeval-grp

34

The "fold-right" pattern

```
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)
foldr (:) [] [1,2,3]
 ==> (:) 1 (foldr (:) [] [2, 3])
  ==> (:) 1 ((:) 2 (foldr (:) [] [3]))
  ==> (:) 1 ((:) 2 ((:) 3 (foldr (:) [] [])))
  ==> (:) 1 ((:) 2 ((:) 3 []))
  == 1 : (2 : (3 : []))
  == [1,2,3]
```

The "fold-right" pattern

```
foldr f b [x1, x2, x3, x4]

==> f x1 (foldr f b [x2, x3, x4])

==> f x1 (f x2 (foldr f b [x3, x4]))

==> f x1 (f x2 (foldr f b [x4])))

==> f x1 (f x2 (f x3 (foldr f b [x4])))

==> f x1 (f x2 (f x3 (f x4 (foldr f b []))))

Accumulate the values from the right

For example:

foldr (+) 0 [1, 2, 3, 4]

==> 1 + (foldr (+) 1 [2, 3, 4])

==> 1 + (2 + (foldr (+) 0 [3, 4]))

==> 1 + (2 + (3 + (foldr (+) 0 [4])))

==> 1 + (2 + (3 + (4 + (foldr (+) 0 []))))

==> 1 + (2 + (3 + (4 + 0)))
```

37

QUIZ

What is the most general type of foldr? *

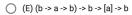
```
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)
```

(A) (a -> a -> a) -> a -> [a] -> a

(B) (a -> a -> b) -> a -> [a] -> b

(C) (a -> b -> a) -> b -> [a] -> b

 $\bigcirc \ \ (D) \ (a -> b -> b) -> b -> [a] -> b$





http://tiny.cc/cse116-foldtype-ind

38

QUIZ

What is the most general type of foldr? *

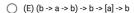
```
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)
```

(A) (a -> a -> a) -> a -> [a] -> a

(B) (a -> a -> b) -> a -> [a] -> b

 \bigcirc (C) (a -> b -> a) -> b -> [a] -> b

(D) (a -> b -> b) -> b -> [a] -> b





http://tiny.cc/cse116-foldtype-grp

The "fold-right" pattern

Is foldr tail recursive?

Answer: No! It calls the binary operations on the results of the recursive call

40

What about tail-recursive versions?

```
Let's write tail-recursive sum!
sumTR :: [Int] -> Int
sumTR = ...
```

41

What about tail-recursive versions?

```
Let's write tail-recursive sum!

sumTR :: [Int] -> Int

sumTR xs = helper 0 xs

where

helper acc [] = acc

helper acc (x:xs) = helper (acc + x) xs
```

What about tail-recursive versions?

```
Lets run sumTR to see how it works
```

Note: helper directly returns the result of recursive call!

43

What about tail-recursive versions?

```
Let's write tail-recursive cat!
catTR :: [String] -> String
catTR = ...
```

44

What about tail-recursive versions?

```
catTR :: [String] -> String
catTR xs = helper "" xs
  where
  helper acc [] = acc
  helper acc (x:xs) = helper (acc ++ x) xs
```

Let's write tail-recursive cat!

What about tail-recursive versions?

Lets run catTR to see how it works

Note: helper directly returns the result of recursive call!

46

Can you spot the pattern?

The "fold-left" pattern



The fold1 Pattern

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

The "fold-left" pattern

49

QUIZ

http://tiny.cc/cse116-foldl-ind

50

QUIZ

http://tiny.cc/cse116-foldl-grp

QUIZ

QUIZ

The "fold-left" pattern

```
foldl f b
                                   [x1, x2, x3, x4]
  ==> helper b
                                   [x1, x2, x3, x4]
  ==> helper (f b x1)
                                        [x2, x3, x4]
  ==> helper (f (f b x1) x2)
==> helper (f (f (f b x1) x2) x3)
  ==> helper (f (f (f (f b x1) x2) x3) x4) []
  ==> (f (f (f (f b x1) x2) x3) x4)
Accumulate the values from the left
For example:
foldl (+) 0
                                   [1, 2, 3, 4]
[1, 2, 3, 4]
  ==> helper 0
  ==> helper (0 + 1)
                                       [2, 3, 4]
  ==> helper ((0 + 1) + 2)
  ==> helper (((0 + 1) + 2) + 3) [4]
==> helper (((0 + 1) + 2) + 3) + 4) []
  ==>((((0+1)+2)+3)+4)
```

Left vs. Right

```
foldl f b [x1, x2, x3] ==> f (f (f b x1) x2) x3 -- Left

foldr f b [x1, x2, x3] ==> f x1 (f x2 (f x3 b)) -- Right

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 -- Left

foldr :: (a -> b -> b) -> b -> [a] -> b -- Right
```

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
```

56

55

Useful HOF: compose

```
-- you can write
map (\x -> f (g x)) ys

-- more concisely like so:
map (f . g) ys
What is the type of (.)?

(.) :: (b -> c) -> (a -> b) -> a -> c
```

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 operation to combine results

Useful helper HOFs:

- Flip the order of function's (first two) arguments
- Compose two functions

58

Higher Order Functions

HOFs can be put into libraries to enable modularity

- Data 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
 - no need to know the implementation of the collection

Enabled the "big data" revolution e.g. MapReduce, Spark

59

That's all folks!