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

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:

```haskell
-- evens []  ==> []
-- evens [1,2,3,4] ==> [2,4]
evens :: [Int] -> [Int]
evens []  = ...
evens (x:xs) = ...
```

Example: four-letter words

Let’s write a function fourChars:

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

Yikes, Most Code is the Same!

```haskell
foo []  = []
foo (x:xs)
 | x mod 2 == 0  = x : foo xs
 | otherwise      = foo xs

foo []  = []
foo (x:xs)
 | length x == 4  = x : foo xs
 | otherwise      = foo xs
```

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?

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

The “filter” pattern

```
evens [] = []
evens (x:xs) | x mod 2 == 0 = x : evens xs
| otherwise = evens xs

fourChars [] = []
fourChars (x:xs) | length x == 4 = x : fourChars xs
| otherwise = fourChars xs

filter f [] = []
filter f (x:xs) | f x = x : filter f xs
| otherwise = filter f xs
```

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

```
filter f []     = []
filter f (x:xs) = x : filter f xs
| otherwise  =  filter f xs
```

```
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 :: ???
```
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 :: ???

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
  • 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]

Example: all caps

Let’s write a function shout:

-- shout [] ==> []
-- shout [‘h’,’e’,’l’,’l’,’o’] ==> [‘H’,’E’,’L’,’L’,’O’]
shout :: [Char] -> [Char]
shout [] = ...
shout (x:xs) = ...
Example: squares

Let's write a function squares:

```haskell
-- squares [] ==> []
-- squares [1,2,3,4] ==> [1,4,9,16]
squares :: [Int] -> [Int]
squares [] = ...
squares (x:xs) = ...
```

Yikes, Most Code is the Same!

Let's rename the functions to foo:

```haskell
-- shout
foo [] = []
foo (x:xs) = toUpper x : foo xs

-- squares
foo [] = []
foo (x:xs) = (x * x) : foo xs

Let's refactor into the common pattern

pattern = ...
```

The “map” pattern

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

The map Pattern

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

Specific Operations
- Transformations toUpper and \( \backslash x \rightarrow x \times x \)
The “map” pattern

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

Lets refactor shout and squares

shout = map ...

squares = map ...

shout = map (\x -> toUpper x)  squares = map (\x -> x*x)

QUIZ

What is the type of map?*

map f [] = []
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/cmps112-map-ind

QUIZ

What is the type of map?*

map f [] = []
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/cmps112-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

\[
\text{map} :: (a \to b) \to [a] \to [b]
\]

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 \( \text{map'} :: (a \to b) \to [a] \to [b] \) whose behavior is different from \( \text{map} \)?

- can you write a function \( \text{map'} :: (a \to b) \to [a] \to [b] \) such that \( \text{map'} f \, xs \) returns a list whose elements are not in \( \text{map} f \, xs \)?

QUIZ

What is the value of quiz? *

\[
\text{map} :: (a \to b) \to [a] \to [b]
\]

\[
\text{quiz} = \text{map} \, ((x, y) \to x + y) \, [1, 2, 3]
\]

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

http://tiny.cc/cmps112-quiz-ind

QUIZ

What is the value of quiz? *

\[
\text{map} :: (a \to b) \to [a] \to [b]
\]

\[
\text{quiz} = \text{map} \, ((x, y) \to x + y) \, [1, 2, 3]
\]

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

http://tiny.cc/cmps112-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

Recall: length of a list

```haskell
-- len [] ==> 0
-- len ["carne","asada"] ==> 2
len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

Recall: summing a list

```haskell
-- 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`:

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

cat :: [String] -> String

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

Can you spot the pattern?

```haskell
-- 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 = ...
```

The “fold-right” pattern

<table>
<thead>
<tr>
<th>len [] = 0</th>
<th>sum [] = 0</th>
<th>cat [] = &quot;&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>len (x:xs) = 1 + len xs</td>
<td>sum (x:xs) = x + sum xs</td>
<td>cat (x:xs) = x ++ sum xs</td>
</tr>
</tbody>
</table>

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

The foldr Pattern

General Pattern

- Recurse on tail
- Combine result with the head using some binary operation
**The “fold-right” pattern**

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

Let's refactor `sum`, `len` and `cat`:

```haskell
sum = foldr ... ...

len = foldr ... ...

len = foldr ... ...
```

Factor the recursion out!

---

You can write it more clearly as

```haskell
sum = foldr (+) 0

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

---

You can write it more clearly as

```haskell
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

```haskell
sum = foldr (+) 0

cat = foldr (++) ""
```
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 (::) [] [3]))
=> (::) 1 (::) 2 (::) 3 [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 (f x3 (foldr f b [x4])))
==⇒ f x1 (f x2 (f x3 (f x4 (foldr f b []))))
==⇒ f x1 (f x2 (f x3 (f x4 b)))

Accumulate the values from the right

For example:

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

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

http://tiny.cc/cmps112-foldtype-ind

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

http://tiny.cc/cmps112-foldtype-grp
The “fold-right” pattern

Is `foldr` tail recursive?

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

What about tail-recursive versions?

Let’s write tail-recursive `sum`!

```haskell
sumTR :: [Int] -> Int
sumTR = ...
```

What about tail-recursive versions?

Let’s write tail-recursive `sum`!

```haskell
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?

Let's run `sumTR` to see how it works:

```haskell
sumTR [1,2,3]
    ==> helper 0 [1,2,3]
    ==> helper 1 [2,3]  -- 0 + 1 ==> 1
    ==> helper 3 [ ]    -- 1 + 2 ==> 3
    ==> helper 6 [ ]    -- 3 + 3 ==> 6
    ==> 6
```

Note: `helper` directly returns the result of recursive call!

What about tail-recursive versions?

Let's write tail-recursive `cat`!

```haskell
catTR :: [String] -> String
catTR xs = helper "" xs

where
    helper acc []  = acc
    helper acc (x:xs) = helper (acc ++ x) xs
```

```haskell
catTR :: [String] -> String
catTR = ...
```
What about tail-recursive versions?

Let's run `catTR` to see how it works:

```
let catTR = [
  "carne", "asada", "torta"
] => helper "" [
  "carne", "asada", "torta"
] => helper "carne" [
  "asada", "torta"
] => helper "carneasada" [
  "torta"
] => helper "carneasadatorta" [] => "carneasadatorta"
```

Note: `helper` directly returns the result of recursive call!

Can you spot the pattern?

```
-- sumTR
foo xs = helper 0 xs
    where
      helper acc [] = acc
      helper acc (x:xs) = helper (acc + x) xs

-- catTR
foo xs = helper "" xs
    where
      helper acc [] = acc
      helper acc (x:xs) = helper (acc ++ x) xs

pattern = ...
```

The “fold-left” pattern

```
sum xs = helper 0 xs
    where
      helper acc [] = acc
      helper acc (x:xs) = helper (acc + x) xs

cat xs = helper "" xs
    where
      helper acc [] = acc
      helper acc (x:xs) = helper (acc ++ x) xs

fold f (b x) = helper b xs
    where
      helper acc [] = acc
      helper acc (x:xs) = helper (f acc x) xs
```

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

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!

QUIZ

What does this evaluate to?*

foldl f b xs = helper b xs
where
  helper acc [] = acc
  helper acc (x:xs) = helper (f acc x) xs

quiz = foldl (:) [] [1,2,3]

○ (A) Type error
○ (B) [1,2,3]
○ (C) [1,2,1]
○ (D) [1,2,1,1]
○ (E) [1,2,1,1,1]

QUIZ

What does this evaluate to?*

foldl f b xs = helper b xs
where
  helper acc [] = acc
  helper acc (x:xs) = helper (f acc x) xs

quiz = foldl (:) [] [1,2,3]

○ (A) Type error
○ (B) [1,2,3]
○ (C) [1,2,1]
○ (D) [1,2,1,1]
○ (E) [1,2,1,1,1]
The “fold-left” pattern

fold1 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) [x3, x4]
  == helper (f (f (f b x1) x2) x3) [x4]
  == 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:

fold1 (+) 0 [1, 2, 3, 4]
  == helper 0 [1, 2, 3, 4]
  == helper (0 + 1) [2, 3, 4]
  == helper (((0 + 1) + 2) [3, 4]
  == helper ((((0 + 1) + 2) + 3) [4]
  == helper ((((0 + 1) + 2) + 3) + 4) []
  == (((0 + 1) + 2) + 3) + 4)
**Left vs. Right**

\[\text{foldl} \ f \ b \ [x_1, x_2, x_3] \implies f \ (f \ (f \ b \ x_1) \ x_2) \ x_3 \quad \text{-- Left}\]

\[\text{foldr} \ f \ b \ [x_1, x_2, x_3] \implies f \ x_1 \ (f \ x_2 \ (f \ x_3 \ b)) \quad \text{-- Right}\]

For example:

\[\text{foldl} \ (\cdot) \ 0 \ [1, 2, 3] \implies ((0 + 1) + 2) + 3 \quad \text{-- Left}\]

\[\text{foldr} \ (\cdot) \ 0 \ [1, 2, 3] \implies 1 + (2 + (3 + 0)) \quad \text{-- Right}\]

Different types!

\[\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \quad \text{-- Left}\]

\[\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \quad \text{-- Right}\]

**Useful HOF: flip**

-- you can write

\[\text{foldl} \ (\lambda \ x s \rightarrow x : xs) \ [] \ [1,2,3]\]

-- more concisely like so:

\[\text{foldl} \ (\text{flip} \ () \ ) \ [] \ [1,2,3]\]

What is the type of flip?

\[\text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c\]

**Useful HOF: compose**

-- you can write

\[\text{map} \ (\lambda \ x \rightarrow f \ (g \ x)) \ ys\]

-- more concisely like so:

\[\text{map} \ (f \ . \ g) \ ys\]

What is the type of (\ . )?

\[\ (.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c\]
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

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) -\rightarrow 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

That’s all folks!