Higher-Order Functions

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:
-- 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:

-- fourChars [] ==> []
-- fourChars ["i","must","do","work"] ==> ["must","work"]

fourChars :: [String] -> [String]
fourChars [] = ...
fourChars (x:xs) = ...
Yikes, Most Code is the Same

Lets rename the functions to `foo`:

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

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

Only difference is condition
• $x \mod 2 = 0$ vs $\text{length } x = 4$

**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

```haskell
filter (\x -> x `mod` 2 == 0)
```

```
filter (\x -> length x == 4)
```

The filter Pattern

General Pattern

- HOF filter
- Recursively traverse list and pick out elements that satisfy a predicate

Specific Operations

- 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

filter instances

Avoid duplicating code!
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 :: ???

-- 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 :: ???
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

Lets write a function `shout`:

```haskell
-- shout []        ==> []
-- shout ['h','e','l','l','o']  ==> ['H','E','L','L','O']

shout :: [Char] -> [Char]
shout []    = ...
shout (x:xs) = ...
```
Example: squares

Lets 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

Lets rename the functions to foo:
-- 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

General Pattern

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

Specific Operations

- Transformations toUpper and \( \lambda x \rightarrow x \ast x \)

\[
\begin{align*}
\text{shout} \; [] & = [] \\
\text{shout} \; (x:xs) & = \text{toUpper} \; \! x \; : \; \text{shout} \; xs \\
\text{squares} \; [] & = [] \\
\text{squares} \; (x:xs) & = (x \ast x) \; : \; \text{squares} \; xs \\
\text{map} \; f \; [] & = [] \\
\text{map} \; f \; (x:xs) & = f \; x \; : \; \text{map} \; f \; xs
\end{align*}
\]

Let's refactor shout and squares
shout  = map ... 

squares = map ...

\[
\begin{align*}
\text{map } f \; [ ] & \; = \; [ ] \\
\text{map } f \; (x:xs) & \; = \; f \; x \; : \; \text{map } f \; xs
\end{align*}
\]

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

map instances
QUIZ

What is the type of \texttt{map}?

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

(A) (Char -> Char) -> [Char] -> [Char]  \xmark

(B) (Int -> Int) -> [Int] -> [Int]  \xmark

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

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

(E) (a -> b) -> [c] -> [d]
F o r a n y t y p e s `a` a n d `b`
if you give me a t r a n s f o r m a t i o n f r o m `a` t o `b`
and a l i s t o f `a`'s,
I'll give you b a c k a l i s t o f `b`'s

map :: (a -> b) -> [a] -> [b]

T y p e s a y s i t a l l!

- The o n l y m e a n i n g f u l t h i n g a f u n c t i o n o f t h i s t y p e c a n d o is a p p l y i t s f i r s t a r g e m e n t t o e l e m e n t s o f t h e l i s t
- H o o g l e i t!

T h i n g s t o t r y a t h o m e:

- c a n y o u w r i t e a f u n c t i o n map' :: (a -> b) -> [a] -> [b] whose b e h a v i o r i s d i f f e r e n t f r o m map?
- c a n y o u w r i t e a f u n c t i o n map' :: (a -> b) -> [a] -> [b] s u c h t h a t map' f xs r e t u r n s a l i s t w h o s e e l e m e n t s a r e n o t i n map f xs?
QUIZ

What is the value of quiz?

map :: (a -> b) -> [a] -> [b]

\[(\text{int, int}) \rightarrow \text{int}\]

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

(A) [2, 4, 6]

(B) [3, 5]

(C) Syntax Error

(D) Type Error

(E) None of the above
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

-- len [] ==> 0
-- len ["carne","asada"] ==> 2

len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
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) = ...
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 = ...
The “fold-right” pattern

\[
\begin{align*}
\text{len} & \colon \mathbb{N} & \text{len} \ [\ ] & = 0 \\
& & \text{len} \ (x \mathbin{:} xs) & = 1 + \text{len} \ xs \\
\text{sum} & \colon \mathbb{N} & \text{sum} \ [\ ] & = 0 \\
& & \text{sum} \ (x \mathbin{:} xs) & = x + \text{sum} \ xs \\
\text{cat} & \colon \mathbb{N} & \text{cat} \ [\ ] & = "" \\
& & \text{cat} \ (x \mathbin{:} xs) & = x \ ++ \ \text{sum} \ xs
\end{align*}
\]

\[
\begin{align*}
\text{foldr} \ f \ b \ [\ ] & = b \\
\text{foldr} \ f \ b \ (x \mathbin{:} xs) & = f \ x \ (\text{foldr} \ f \ b \ xs)
\end{align*}
\]

The \text{foldr} Pattern

General Pattern

- Recurse on tail
- Combine result with the head using some binary operation
foldr \( f \ b \ [ \] \) = b
foldr \( f \ b \ (x:xs) \) = \( f \ x \ (\text{foldr} \ f \ b \ xs) \)

Let’s refactor \text{sum}, \text{len} \text{ and } \text{cat}:

\text{sum} = \text{foldr} \ldots \ldots
\text{cat} = \text{foldr} \ldots \ldots
\text{len} = \text{foldr} \ldots \ldots

Factor the recursion out!
foldr \( f \ b \ [\] \) = \( b \)  
foldr \( f \ b \ (x:xs) \) = \( f \ x \ (\text{foldr} \ f \ b \ xs) \)

\[
\begin{align*}
\text{len} &= \text{foldr} \ (\lambda x \ n \rightarrow 1 + n) \ 0 \\
\text{sum} &= \text{foldr} \ (\lambda x \ n \rightarrow x + n) \ 0 \\
\text{cat} &= \text{foldr} \ (\lambda x \ s \rightarrow x ++ n) \ \"\" \\
\end{align*}
\]

foldr instances

You can write it more clearly as

\[
\begin{align*}
\text{sum} &= \text{foldr} \ (+) \ 0 \\
\text{cat} &= \text{foldr} \ (++) \ "" \\
\end{align*}
\]
**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]

(D) [[3],[2],[1]]

(E) [[1],[2],[3]]
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 (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 (+) 0 [3, 4]))
  ==> 1 + (2 + (3 + (foldr (+) 0 [4])))
  ==> 1 + (2 + (3 + (4 + (foldr (+) 0 []))))
  ==> 1 + (2 + (3 + (4 + 0)))
**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`
Is foldr tail-recursive?

What about tail-recursive versions?
Let's write tail-recursive sum!

\[
\text{sumTR} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sumTR} = \ldots
\]

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

\[
\text{sumTR} \ [1,2,3] \\
\quad \Rightarrow \text{helper 0} \ [1,2,3] \\
\quad \Rightarrow \text{helper 1} \ [2,3] \quad -- \ 0 + 1 \Rightarrow 1 \\
\quad \Rightarrow \text{helper 3} \ [3] \quad -- \ 1 + 2 \Rightarrow 3 \\
\quad \Rightarrow \text{helper 6} \ [\ ] \quad -- \ 3 + 3 \Rightarrow 6 \\
\quad \Rightarrow 6
\]

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