ANN: Midterm on FRIDAY

Higher-Order Functions

HW3 - Fold

Plan for this week

each-of "record"

one-of "union" or "sum"

recursive $\boxed{\text{List} \; \boxed{\text{List}}}$

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

Lets rename the functions to `foo`:

```
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

filter (\x -> x 'mod' 2 == 0)

The "filter" pattern

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

```
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
```

The `filter` Pattern

General Pattern

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

Specific Operations

- Predicates `isEven` and `isFour`

$\lambda x.\ x+1$

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

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

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

`filter` instances

Avoid duplicating code!

Let's talk about types

```haskell
-- 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 :: ???
```

```haskell
-- 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`:

```
-- 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` :

```
-- 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` :

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

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

Lets **refactor** into the **common pattern**

```haskell
pattern = ...
```

The “map” pattern

```
shout []     = []
shout (x:xs) = toUpper x : shout xs
```

```
squares []     = []
squares (x:xs) = (x*x) : squares xs
```

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

The `map` Pattern

General Pattern

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

Specific Operations

- Transformations `toUpper` and `\x -> x * x`

```
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)
```

`map` instances

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]`

```haskell
-- 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`?

QUIZ

What is the value of `quiz` ?

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

(Int, Int) → Int

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

[Int]

(A) `[2, 4, 6]`

(B) `[3, 5]`

(C) Syntax Error

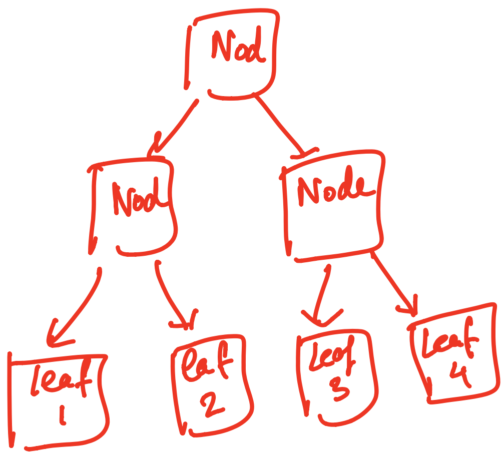
(D) Type Error

(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

```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?

```
-- 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
```

Base

OP

```
pattern = ...
```

The "fold-right" pattern

```
len []     = 0
len (x:xs) = 1 + len xs
```

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

```
cat []     = “”
cat (x:xs) = x ++ sum xs
```

```
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

```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 ...  ...

cat = foldr ...  ...

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

Factor the recursion out!

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

```haskell
len = foldr (\x n -> 1 + n) 0
```

```haskell
sum = foldr (\x n -> x + n) 0
```

```haskell
cat = foldr (\x s -> x ++ n) “”
```

`foldr` instances

You can write it more clearly as

```haskell
sum = foldr (+) 0
```

```haskell
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]`

(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]
```

$(((b \text{ op } x_1) \text{ op } x_2) \text{ op } x_3) \text{ op } x_4)$

foldl

$((((x_1 \text{ op } x_2) \text{ op } x_3) \text{ op } x_4) \text{ op } b)$

$(x_1 \text{ op } (x_2 \text{ op } (x_3 \text{ op } (x_4 \text{ op } b))))$

foldr

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`!

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

Lets run `sumTR` to see how it works

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

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