# Higher-Order Functions; Recursion Patterns on Lists; Higher-Order Programming (slides, where noted, due to Graham Hutton)

## Announcements

- We will have Quiz 1 on Friday
- I'll post PS #3 sometimes after class





## Outline

- Higher-order functions
- Recursion patterns on lists
  - map, filter and fold
- Function composition and partial application
- "Wholemeal" programming
- Other useful higher-order Prelude functions





## Higher-Order Functions

A function is called higher-order if it takes a function as an argument or returns a function as a result

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

## Why Are They Useful?

- Common programming idioms can be encoded as functions within the language itself
- Domain specific languages can be defined as collections of higher-order functions
- Algebraic properties of higher-order functions can be used to reason about programs

Programming in Haskell, slide due to G. Hutton 5

5

## Things to Do with a List

- Run some function on every element of the list *map*
- "Filter" according to a predicate: keep only certain elements of the list, filter out rest *filter*
- "Reduce" a list into a value in some way, e.g., find maximal value, sum, product, etc. *fold*
- Other?

Programming in Haskell, A Milanova 6

6

## The Map Function

The higher-order library function called map applies a function to every element of a list

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

For example:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

Programming in Haskell, slide due to G. Hutton 7

7

```
> map f xs
```

```
[ e1,  e2,  e3, …  en ]
   f    f    f      f
[ r1,  r2,  r3, …  rn ]
```

There is a build-in function map in Prelude

Programming in Haskell, A Milanova 8

8

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x ← xs]
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

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





## The Filter Function

The higher-order library function filter selects every element from a list that satisfies a predicate

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]

[2,4,6,8,10]
```





Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using recursion:

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





## The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to some value v, and any non-empty list to some function ⊕ applied to its head and f of its tail.

```
f []     = v
f (x:xs) = x ⊕ f xs
```

```
[   x1 ,   …   xn-1 ,   xn    ]     v
[   x1 ,   …   xn-1   ]   res1
  …
[   x1 ]   resn-1

  resn
```

13

13

For example:

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

V = 0
⊕ = +

```
product []     = 1
product (x:xs) = x * product xs
```

V = 1
⊕ = *

```
and []     = True
and (x:xs) = x && and xs
```

V = True
⊕ = &&

Programming in Haskell, slide due to G. Hutton 14

14

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function ⊕ and the value v as arguments.

For example:

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

product xs = foldr (*) 1 xs

or xs = foldr (||) False xs

and xs = foldr (&&) True xs
```

Programming in Haskell, slide due to G. Hutton 15

15

Foldr itself can be defined using recursion:

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

However, it is best to think of foldr non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

Programming in Haskell, slide due to G. Hutton 16

16

For example:

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  1+(2+(3+0))
=
  6
```

Replace each (:) by (+) and [] by 0.

Programming in Haskell, slide due to G. Hutton 17

17

For example:

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
  1*(2*(3*1))
=
  6
```

Replace each (:) by (*) and [] by 1.

Programming in Haskell, slide due to G. Hutton 18

18

## Other Foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

length xs = foldr (\elem res -> 1+res) 0 xs

Programming in Haskell, slide due to G. Hutton 19

19

For example:

```
  length [1,2,3]
=
  length (1:(2:(3:[])))
=
  1+(1+(1+0))
=
  3
```

Replace each (:) by _ n -> 1+n and [] by 0.

Hence, we have:

```
length xs = foldr (\_ n → 1+n) 0 xs
```

Programming in Haskell, modified from a slide due to G. Hutton 20

20

Now recall the reverse function:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

For example:

```
  reverse [1,2,3]
=
  reverse (1:(2:(3:[])))
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Replace each (:) by \x xs → xs ++ [x] and [] by [].





Hence, we have:

```
reverse ls = foldr (\x xs → xs ++ [x]) [] ls
```

elem partial result

Finally, we note that the append function (++) has a particularly compact definition using foldr:

```
xs ++ ys = foldr (:) ys xs
```

Replace each (:) by (:) and [] by ys.





## Why Is Foldr Useful?

- Some recursive functions on lists, such as sum, are simpler to define using foldr
- Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as fusion and the banana split rule
- Advanced program optimizations can be simpler if foldr is used in place of explicit recursion





## Exercises

The following data type represents arbitrarily nested lists:

```
data NL a = Atom a
          | List [NL a]
```

```
> a = Atom 1
> x = List [Atom 1, Atom 2, Atom 3]
> atomcount x
3

> y = List [x]
> z = List [ List [ List [Atom 1] ] ]
```

[1, 2, 3]

[[1, 2, 3]]

[[1,2,3],[[1]]]

[[[1]]]

## Exercises

```
data NL a = Atom a
          | List [NL a]
```

Write `atomcount :: Num p => NL a -> p` to count number of atoms in a nested list. Use `map` and `foldr`.

```
atomcount (Atom x) = 1
atomcount (List []) = 0
atomcount (List (x:xs)) = atomcount x +
                          atomcount (List xs)

atomcount (Atom x ) = 1
atomcount (List xs) = foldr (+) 0 (map atomcount xs)

:t atomcount            a@(Atom x)
```





## Exercises

```
data NL a = Atom a
          | List [NL a]
```

Write `flatten :: NL a -> [a]` to flatten a nested list. Use `map` and `foldr`. E.g.,

```
> flatten (List [Atom 0, List [List [Atom 1]]])
[0,1]
```

```
flatten (Atom x) = [x]
flatten (List xs) = foldr (++) [] (map flatten xs)
```

Do you see a common pattern here?





## The foldl Function

foldl "folds" the list, like foldr does, but from left to right:

```
f v (x:xs) = f (v ⊕ x) xs
f v []     = v
```

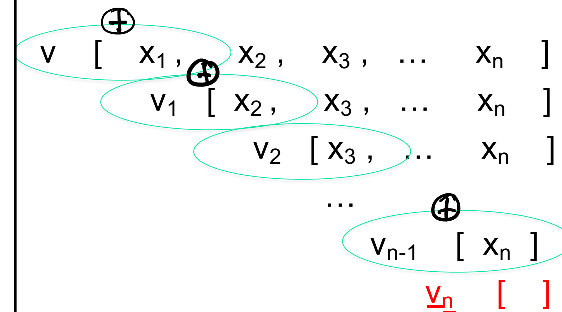
f takes a value v and a list, as well as an implicit third argument ⊕.
A non-empty list maps to f applied on 2 arguments:
arg1 is v ⊕ head-of-list and arg2 is tail-of-list.
f applied on the empty list passes back "accumulated" value v.





## foldl

```
foldl ⊕ v [x₁,x₂,x₃] = ((v ⊕ x₁) ⊕ x2) ⊕ x3
```



For example:

```
sum [1,2,3]
=
foldl (+) 0 [1,2,3]
=
foldl (+) 0 (1:(2:(3:[])))
=
(((0+1)+2)+3)
=
6
```

We cannot replace each (:) by (+) as we did with foldr.





For example:

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

```
foldl (-) 0 [1,2,3]
=
foldl (-) 0 (1:(2:(3:[])))
=
(((0-1)-2)-3)
=
-6
```





# Other Foldl Examples

```
length' :: [a] → Int
length' xs = foldl  ?  ?  xs
```

```
length' xs = foldl (\n _ → 1+n) 0 xs
```

versus:

```
length' xs = foldr (\_ n → 1+n) 0 xs
```





# Other Foldl Examples

```
reverse' :: [a] → Int
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' xs = foldl (\v el → [el]++v) [] xs
```

el:v

versus

```
reverse' xs = foldr (\el v → v++[el]) [] xs
```

Note: use foldl' from Data.List.

## foldl vs. foldr vs. foldl'

```
Prelude Data.List> :set +s
Prelude Data.List> foldl (+) 0 [0..1000000]
500000500000
(0.23 secs, 161,298,112 bytes)

Prelude Data.List> foldr (+) 0 [0..1000000]
500000500000
(0.17 secs, 161,588,432 bytes)

Prelude Data.List> foldl' (+) 0 [0..1000000]
500000500000
(0.05 secs, 88,071,320 bytes)
```

Why?





## Outline

- Higher-order functions
- Recursion patterns
  - map, filter and fold
- Function composition and partial application
- "Wholemeal" programming
- Other useful higher-order prelude functions





## Composition

Can you write a function that has the following type:

```
? :: (b → c) → (a → b) → (a → c)
```

```
foo :: (b → c) → (a → b) → (a → c)
foo f g = \x -> f (g x)
```





The library function (.) returns the composition of two functions as a single function.

```
(.) :: (b → c) → (a → b) → (a → c)
f . g = \x → f (g x)
```

```
f (g (h (i (j (k x))))) == (f.g.h.i.j.k) x
```

For example:

```
odd n :: Int → Bool
odd n = not (even n)
```

```
odd' :: Int → Bool
odd' = not . even
```

```
> :t (.)
(b → c) → (a → b) → (a → c)
```

Why parentheses stay around b → c and a → b but disappear around a → c?

Two versions of foo actually:

```
foo :: (b → c) → (a → b) → (a → c)
foo f g = \x -> f (g x)
```

```
foo f g x = f (g x)
```

fun x = expr is syntactic sugar fun = \x→ expr





## Partial Application

Partial application is common in Haskell:

```
foo f g x = f (g x)
```

```
compFandG = foo f g
```

fun x = expr or fun = \x→ expr

Other related terms for this style are eta-reduction/expansion, pointfree programming style.





Going back to some definitions we saw earlier:

```
length xs = foldr (\_ n → 1+n) 0 xs
```

Haskell's way is to use higher-order functions:

```
length = foldr (\_ n → 1+n) 0
```

We define length as partial application of foldr.





## "Wholemeal" Programming

- Haskell's way
- Think in terms of whole lists instead of individual indices
- Use partial application and composition to express complex computations in a compact way

What does foo do? What issues do you see?

```
foo :: [Int]->Int
foo [] = 0
foo (x:xs)
  | x `mod` 3 == 0 = x^3 + foo xs
  | otherwise = foo xs
```

And Haskell's way is…

```
cube x = x^3
div3 x = x `mod` 3 == 0
foo' = foldr (+) 0 . map cube . filter div3
```

point free    partial application



Some notes:

- Fusion law for map: `map f . map g = map (f.g)`
- Can you express a general list comprehension `[ f x | x <- xs, p x ]` in terms of map and filter?



Some notes:

- `($) :: (a -> b) -> a -> b` changes order of operations and avoids parentheses. E.g.,

```
foo' xs = sum (map cube (filter div3 xs))
```

```
foo' xs = sum $ map cube $ filter div3 xs
```

foo f g x = f (g x)
= f $ g x



# Exercise

Write inner product in a pointfree style.

E.g., following John Backus's definition in FP:
def IP = (Insert +) (ApplyToAll *) Transpose

```
ip [[x1,x2,...,xn],[y1,y2,...,yn]] ==
  x1*y1 + x2*y2 + … xn*yn
```

Notes:

Pointfree is NOT always a good thing...
lambdabot in the #haskell IRC channel has a command @pl for turning functions into equivalent pointfree expressions. Here's an example:

```
@pl \f g x y -> f (x ++ g x) (g y)
join . ((flip . ((.) .)) .) . (. ap (++)) . (.)
```

Simplicity and readability!!!



# Outline

- Higher-order functions
- Recursion patterns
  - map, filter and fold
- Function composition and partial application
- "Wholemeal" programming
- Useful higher-order Prelude functions



The library function all decides if every element of a list satisfies a given predicate.

```
all :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

For example:

```
> all even [2,4,6,8,10]

? TRUE
```



Dually, the library function any decides if at least one element of a list satisfies a predicate.

```
any :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

For example:

```
> any (== ' ') "abc def"

? TRUE
```

The library function takeWhile selects elements from a list while a predicate holds of all the elements.

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p [] = []
takeWhile p (x:xs)
    | p x        = x : takeWhile p xs
    | otherwise = []
```

For example:

```
> takeWhile (/= ' ') "abc def"
?abc
```





Dually, the function dropWhile removes elements while a predicate holds of all the elements.

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p [] = []
dropWhile p (x:xs)
    | p x        = dropWhile p xs
    | otherwise = x:xs
```

For example:

```
> dropWhile (== ' ') "   abc"
?abc
```





The function iterate generates an infinite list by applying a function f repeatedly starting from an initial value x:

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

```
iterate f x == [x, f x, f (f x), f (f (f x)) … ]
```

For example:

```
> take 10 $ iterate (+1) 0
?
```





## Exercise

[1, (improve 1) ... ]
$y_0$ $y_1$ $y_2$ $y_3$ ...

Newton's method for finding positive `sqrt(x)`:

```
root :: Float
root x = rootiter x 1

rootiter :: Float -> Float -> Float
rootiter x y
   | satisfactory x y = y
   | otherwise rootiter x (improve x y)

satisfactory x y = abs (y * y - x) < 0.01

improve x y = (y + x/y) / 2
```

root x = head $ filter (satisfactory x) $ iterate (improve x) 1

## Exercises

(1) Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

(2) Redefine map f and filter p using foldr.



## Exercises

(3) Voting. Write a program in that determines the winner of a vote. E.g., in the list of votes below 5 wins:

```
votes = [Int]
votes = [1,2,5,3,1,2,5,6,6,5,5,5,5,4]
```

```
count :: Eq a => a -> [a] -> Int
…
> count 5 votes
6
```

count v = length ∘ filter (== v)



## Exercises

```
rdups :: [a] -> [a]
…
> rdups votes
[1,2,5,3,6,4]
```

rdups [] = []
rdups (x:xs) = x : filter (/=x) (rdups xs)

> summarize votes
[(1,3),(1,4),(2,1),(2,2),(2,6),(6,5)]

summarize votes = sort [(count v votes, v) | v ← rdups votes]

winner votes = snd $ last $ summarize votes



## Moving On To Higher Grounds

In Haskell we recognize and generalize patterns. We already saw patterns on list structures. How about trees?

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

treeSize and treeSum:

```
treeSize :: Tree a → Int
treeSize (Leaf n)    = 1
treeSize (Node x y) = treeSize x + treeSize y

treeSum :: Tree a → Int
treeSum (Leaf n)    = n
treeSum (Node x y) = treeSum x + treeSum y
```

(handwritten: "a" crossed out, "Int" written above; "Num a => Tree a -> a")





treeFlatten and treeDepth:

```
flatten :: Tree a → [a]
flatten (Leaf n)    = [n]
flatten (Node x y) = flatten x ++ flatten y

depth :: Tree a → Int
depth (Leaf n)    = 0
depth (Node x y) = 1 + max (depth x) (depth y)
```





```
data NL a = Atom a
          | List [NL a]
```

And more:

```
atomcount :: NL a → Int
atomcount (Atom n)    = 1
atomcount (List ls) = foldl (+) 0
                        (map atomcount ls)

flatten' :: NL a → [a]
flatten' (Leaf n)    = [n]
flatten' (List ls) = foldl (++) []
                        (map flatten' ls)
```

(handwritten: "Leaf" crossed out, "Atom" written above)





What is the pattern?

- Pattern-match on constructors, Leaf and Node
- In Leaf case (i.e., base case), do a transformation on the a-type value
- In Node case, recurse over each subtree
- Combine results

Generalize! Note: this is not real Haskell code.

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

`foldMap` sketch for `Tree` a datatype:

```
foldMap :: (a -> b) -> Tree a -> b
foldMap f (Leaf n) = f n -- transform
foldMap f (Node l r) = foldMap f l <> -- combine
                       foldMap f r
```



Generalize! Note: this is not real Haskell code.

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

Definition of `foldMap` for `NL` a datatype:

```
foldMap :: (a -> b) -> NL a -> b
foldMap f (Atom n) = f n -- transform
foldMap f (List ls) = foldl <> empty
                          (map (foldMap f) ls)
```



Result type **b** cannot be any type. It must have well-defined `<>` operation and `mempty` value! That is, **b** must be a Monoid type.

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

```
foldMap :: (a -> b) -> NL a -> b
foldMap f (Atom n) = f n -- transform
foldMap f (List ls) = foldl <> mempty
                          (map (foldMap f) ls)
```



# In Mathematics and in Haskell

Magna

Semigroup

```
class Semigroup a where
  (<>) :: a -> a -> a
```

Monoid

```
class Semigroup a => Monoid a where
  mempty :: a
```

Group

More on Semigroup and Monoid type classes next week!