Higher-Order Functions

What is a “Higher Order” Function?

A function which takes another function as a parameter.

E.g. The first argument to map and filter must be a function

Examples

```
Prelude> map even [1, 2, 3, 4, 5]
[False, True, False, True, False]
Prelude> filter even [1, 2, 3, 4, 5]
[2, 4]
```

Why Do We Want Higher-Order Functions?

- Generalise a repeated pattern: define a function to avoid repeating it.
- Higher-order functions let us abstract definitions that are *not exactly the same*, e.g. Use + in one place and * in another
- **Basic idea**: name common code patterns, so we can use them without repeating them

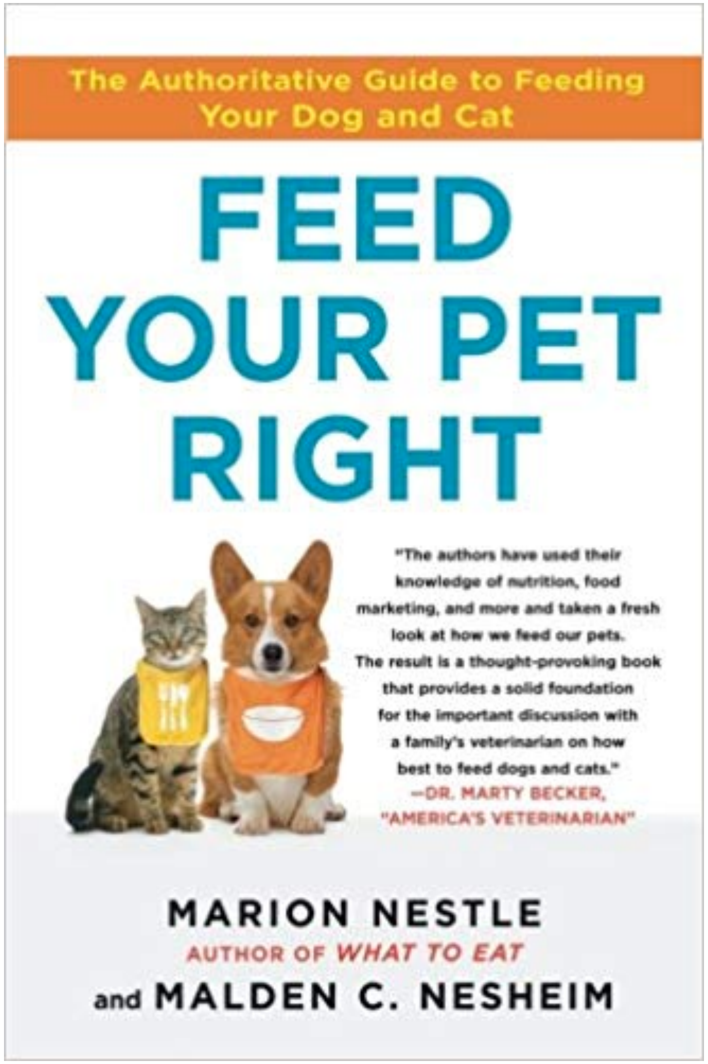
Applications

Combining the elements of a list is a *common* operation.

Now, instead of writing a recursive function, we can just use foldr!

```
product xs       = foldr (*) 1 xs
and xs           = foldr (&&) True xs
concat xs        = foldr (++) [] xs
maximum (x:xs) = foldr max x xs
```

How do we feed Higher-Order Functions



(Back to code)

λ-expressions

```
reverse xs = foldr snoc [] xs
   where snoc y ys = ys++[y]
```

It's a nuisance to need to define snoc, which we only use once! A λ-expression lets us define it where it is used.

```
reverse xs = foldr (λy ys -> ys++[y]) [] xs
```

On the keyboard:

```
reverse xs = foldr (\y ys -> ys++[y]) [] xs
```

Defining unlines

unlines [“abc”, “def”, “ghi”] = “abc\ndef\nghi\n”

unlines [xs,ys,zs] = xs ++ “\n” ++ (ys ++ “\n” ++ (zs ++ “\n” ++ []))

```
unlines xss = foldr (λxs ys -> xs++“\n”++ys) [] xss
```

Just the same as

unlines xss = foldr join [ ] xss

where join xs ys = xs ++ “\n” ++ ys

Further Standard Higher-Order Functions

Function Composition

We can build new functions by composing old functions using **function composition**

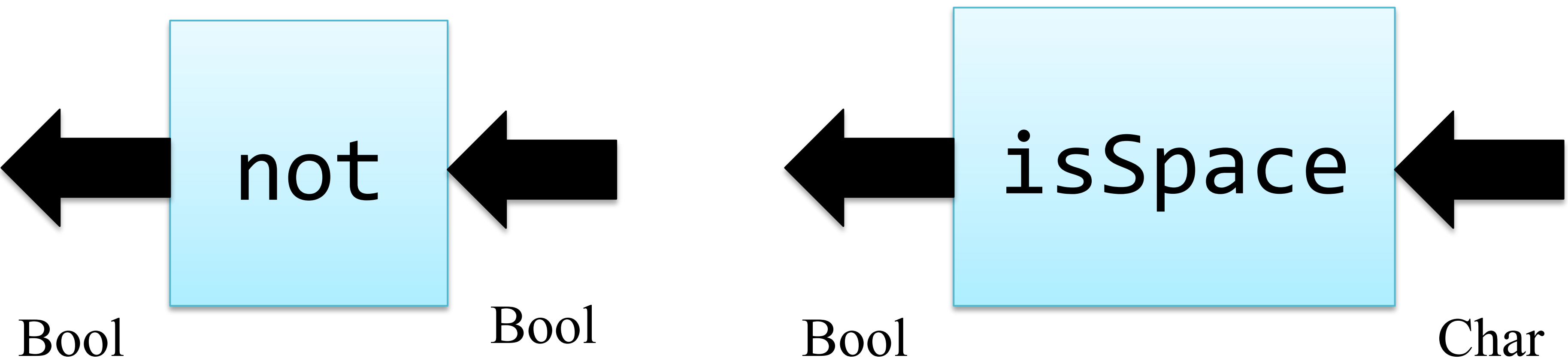
```
notSpace x = not (isSpace x)
```

```
notSpace  =  not . isSpace
```

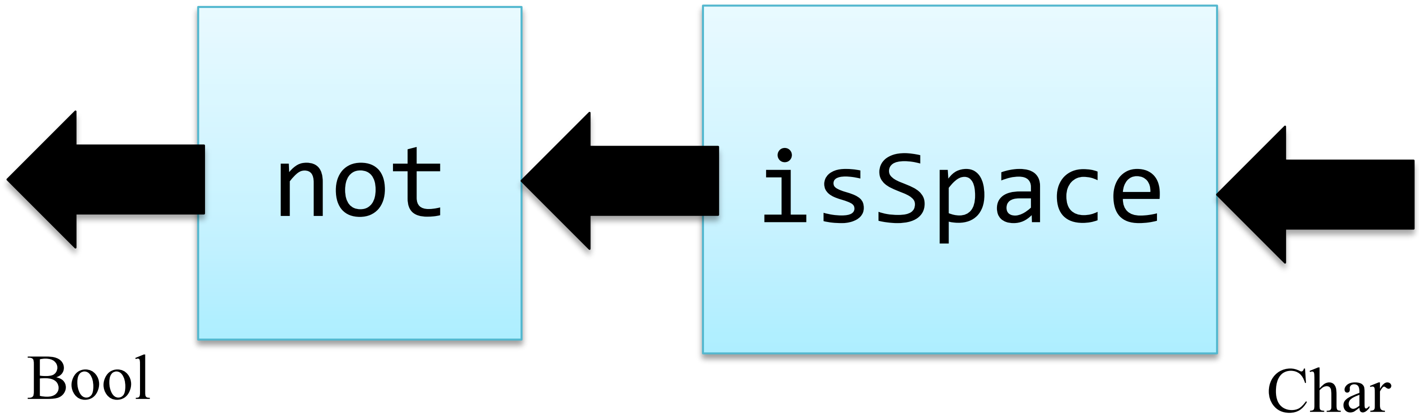
Visually

Visually

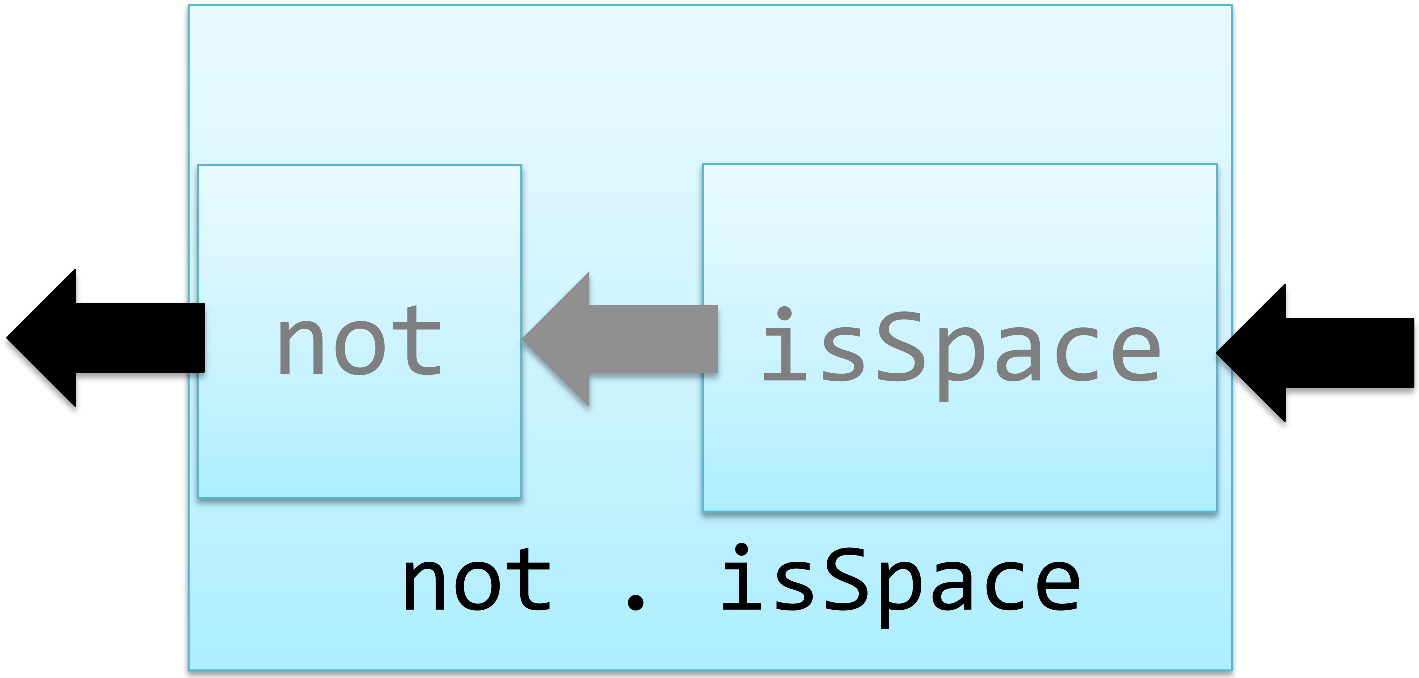
Visually



Visually



Visually



Partial Applications

Haskell has a trick which lets us write down many functions easily.

Insead of

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

Consider this valid alternative definition:

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

foldr is a 3 argument function. It's being called with 2. **What's going on?**

Partial Applications

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

Evaluate `sum [1,2,3]`

= {replacing sum by its definition}

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

= {by the behaviour of foldr}

```
1 + (2 + (3 + 0))
```

= `6`

Now foldr has the *right* number of arguments!

Partial application

```
shout :: String -> String
shout s = map toUpper s
```

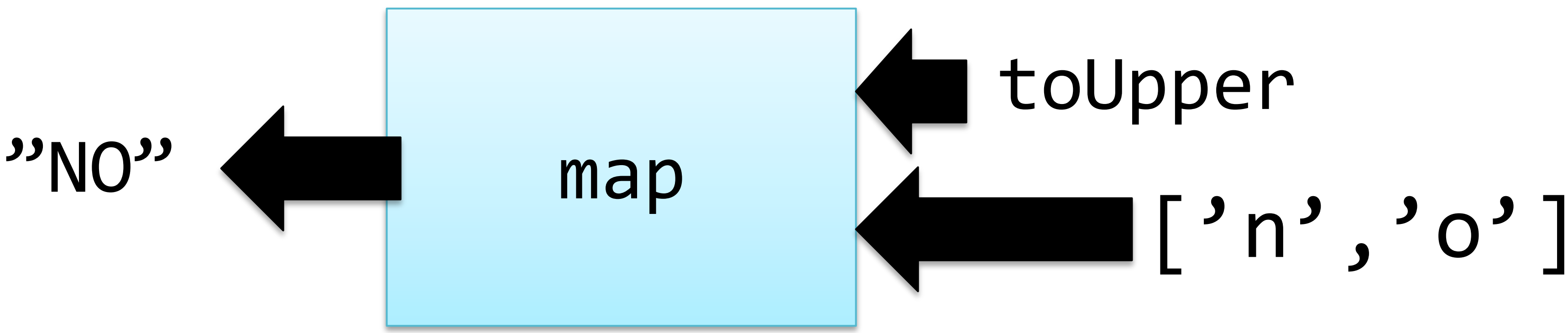
`toUpper :: Char -> Char` lives in `Data.List`
`toUpper ’n’ = ’N’`

Partial application

```
shout :: [Char] -> [Char]
shout s = map toUpper s
```

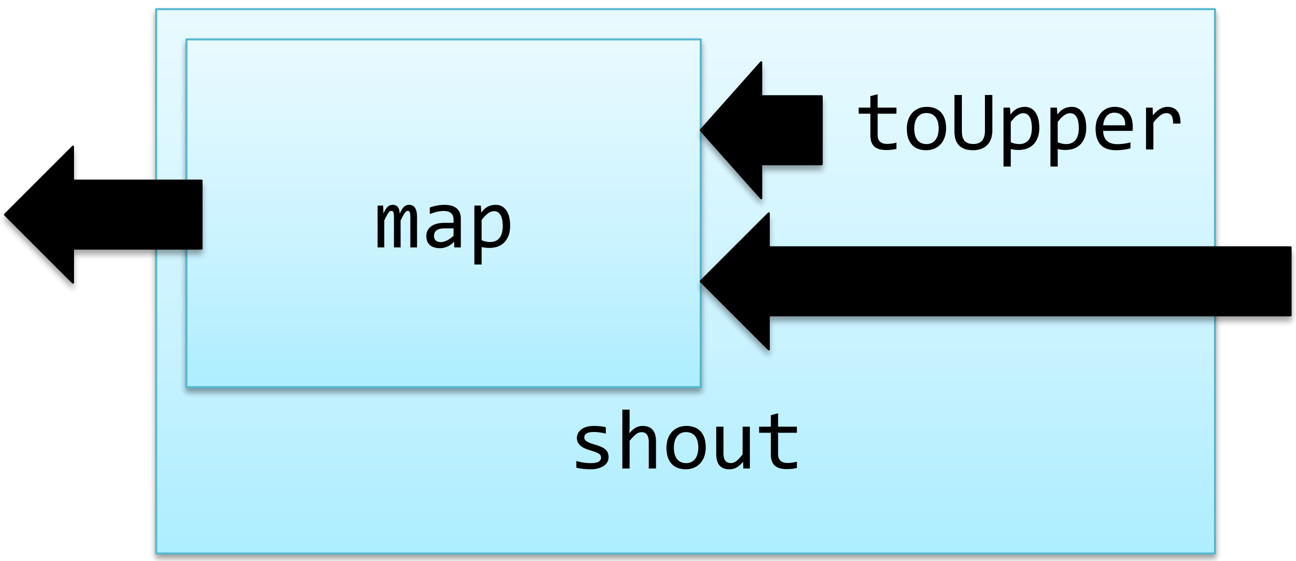
Partial application

```
shout :: [Char] -> [Char]
shout s = map toUpper s
```



Partial application

```
shout :: [Char] -> [Char]
shout = map toUpper
```



Example: combining composition and partial application

The standard function

```
all :: (a -> Bool) -> [a] -> Bool
```

All these are True:

```
all even [2,4,6]
all (<10) [1,2,3]
not (all odd [1,2,3])
```

Example: combining composition and partial application

The standard function

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
```

Example: combining composition and partial application

The standard function

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and (map p xs)
```

Example: combining composition and partial application

The standard function

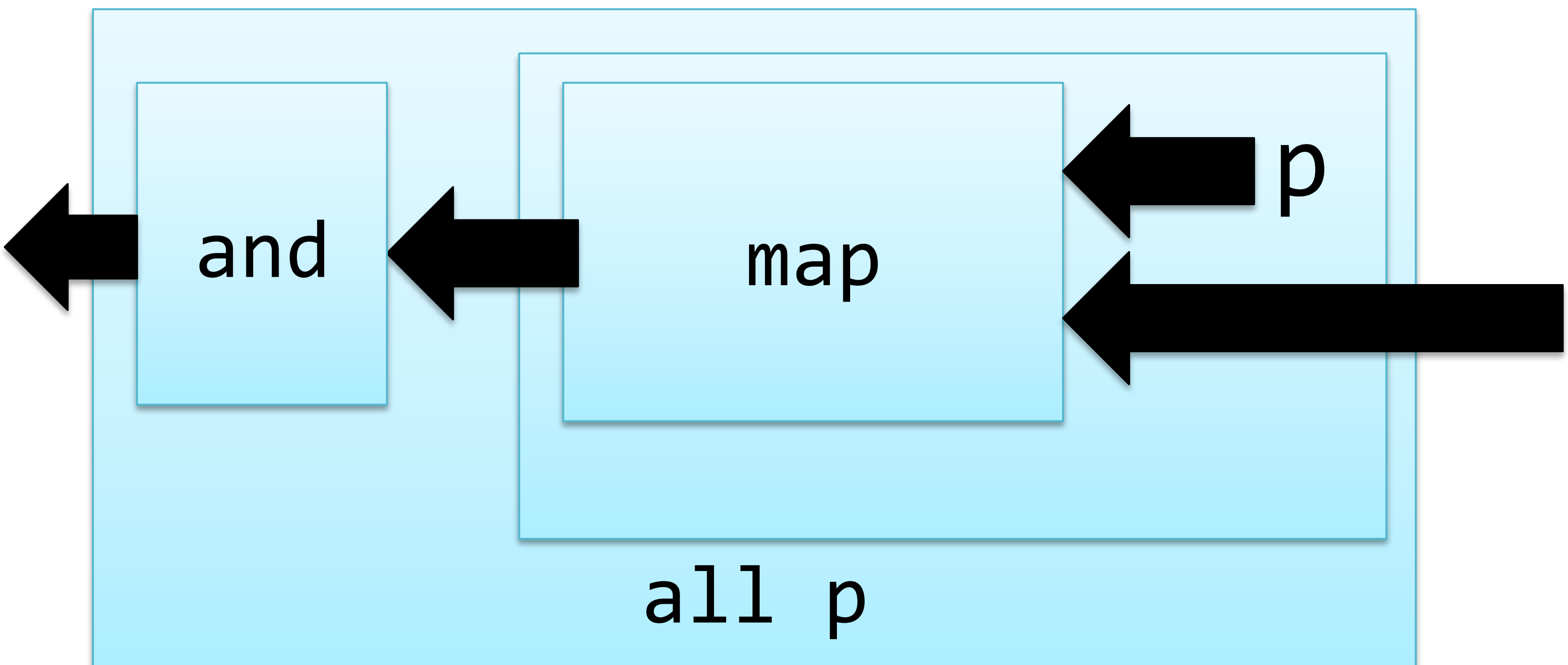
```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and (map p xs)


all p = and . map p
```

A combination of partial application and function composition

Example: combining composition and partial application

The standard function

```
all :: (a -> Bool) -> [a] -> Bool
all p = and . map p
```



Where Do Higher-Order Functions Come From?

- Generalise a repeated pattern: define a function to avoid repeating it.

- Higher-order functions let us abstract patterns that are *not exactly the same*, e.g. Use + in one place and * in another.

- **Basic idea**: name common code patterns, so we can use them without repeating them.

Must I Learn All the Standard Functions?

Yes and No…

- **No**, because they are just defined in Haskell. You can reinvent any you find you need.
- **Yes**, because they capture very frequent patterns; learning them lets you solve many problems with great ease.

”Stand on the shoulders of giants!”

Highlight Rotate Markup Search

PreludeFunctions.pdf (page 1 of 2)

```
:ted functions from the
es: Prelude Data.List
ontrol.Monad
-------------------------
es

 -> Bool

where
 :: a -> a -> Bool
 :: a -> a -> a

> Num a where
a -> a -> a
a -> a
a -> a
Integer -> a

> Real a where
 a -> Rational

 => Integral a where
a -> a -> a
a -> a -> a
a -> Integer

ional a where
a -> a -> a
Rational -> a

> Floating a where
:: a -> a
:: a -> a

nal a) => RealFrac a where
Integral b) => a -> b
(Integral b) => a -> b

-----------------------

ntegral a) => a -> Bool
rem' 2 == 0
 . even

-----------------------

n => [m a] -> m [a]
cons (return [])
 x <- p
 xs <- q
 return (x:xs)
```

```
-- functions on functions
id               :: a -> a
id x             = x

const            :: a -> b -> a
const x _        = x

(.)              :: (b -> c) -> (a -> b) -> a -> c
f . g            = \ x -> f (g x)

flip             :: (a -> b -> c) -> b -> a -> c
flip f x y       = f y x

($)              :: (a -> b) -> a -> b
f $ x            = f x
------------------------------------------
-- functions on Bools
data Bool = False | True

(&&), (||)        :: Bool -> Bool -> Bool
True  && x        = x

not               :: Bool -> Bool
not True          = False
not False         = True
------------------------------------------
-- functions on Maybe
data Maybe a = Nothing | Just a

isJust,isNothing        :: Maybe a -> Bool
isJust (Just a)         =  True
isJust Nothing          =  False

isNothing               =  not . isJust

fromJust                :: Maybe a -> a
fromJust (Just a)       =  a

maybeToList             :: Maybe a -> [a]
maybeToList Nothing     =  []
maybeToList (Just a)    =  [a]

listToMaybe             :: [a] -> Maybe a
listToMaybe []          =  Nothing
listToMaybe (a:_)       =  Just a

catMaybes               :: [Maybe a] -> [a]
catMaybes ls            = [x | Just x <- ls]
------------------------------------------
-- functions on pairs
fst               :: (a,b) -> a
fst (x,y)         =  x
snd               :: (a,b) -> b
snd (x,y)         =  y
```

```
-- functions on lists

map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

head, last          :: [a] -> a
head (x:_)          = x

last [x]            = x

init [x]            = []
init (x:xs)         = x : init xs

null                :: [a] -> Bool
null []             = True
null (_:_)          = False

length              :: [a] -> Int
length              = foldr (const (1+)) 0

(!!)                :: [a] -> Int -> a
(x:_)  !! 0         = x
(_:xs) !! n         = xs !! (n-1)

foldr     :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      =  z
foldr f z (x:xs)  =  f x (foldr f z xs)

foldl     :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      =  z
foldl f z (x:xs)  =  foldl f (f z x) xs

iterate             :: (a -> a) -> a -> [a]
iterate f x         =  x : iterate f (f x)

repeat              :: a -> [a]
repeat x            =  xs where xs = x:xs

replicate           :: Int -> a -> [a]
replicate n x       =  take n (repeat x)
```

See Hompage -> Exam -> PreludeFunctions.pdf

What you should know and use

Operating on the whole of a list:

```
map, filter, (concatMap)
```

Operating on the front of a list

```
takeWhile,  dropWhile
```

Boolean

```
all, any
```

Operating on Pairs

```
zipWith
```

Useful (not essential) but more advanced

Simple useful functions:

```
(.) ($) flip curry uncurry
```

Combining list elements

```
foldr foldl
```

Building lists

```
iterate, groupBy
```

Summary

When to build HOFs

How to feed HOFs

- Named definition
- Lambda expressions
- Sections
- Partial application
- Composition

Lessons

- Higher-order functions take functions as parameters, making them *flexible* and useful in very many situations.
- By writing higher-order functions to capture common patterns, we can reduce the work of programming dramatically.
- λ-expressions, partial applications, function composition and sections help us create functions to pass as parameters, without a separate definition.
- Haskell provides many useful higher-order functions; break problems into small parts, each of which can be solved by an existing function.

Reading

- /learnyouahaskell.com/higher-order-functions