

# Higher-Order Functions

*Original slides by Koen Lindström Claessen*

# What is a “Higher Order” Function?

A function which takes another function as a argument,  
and/or returns a function

## Examples

```
*Main> map even [1,2,3,4,5]  
[False,True,False,True,False]
```

```
*Main> filter even [1,2,3,4,5]  
[2,4]
```

```
even :: Int -> Bool  
even n = n `mod` 2 == 0
```

# What is the Type of filter?

```
*Main> filter even [1,2,3,4,5]  
[2,4]
```

```
even :: Int -> Bool
```

```
filter :: (Int -> Bool) -> [Int] -> [Int]
```

A function type can be  
the type of an argument.

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

# Quiz: What is the Type of map?

## **Example**

```
*Main> map even [1,2,3,4,5]  
[False,True,False,True,False]
```

map also has a polymorphic type -- can you write it down?

# Quiz: What is the Type of map?

## Example

```
*Main> map even [1,2,3,4,5]  
[False,True,False,True,False]
```

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

List of  
results

Any function of  
one argument

Any list of  
arguments

# Quiz: What is the Definition of map?

## Example

```
*Main> map even [1,2,3,4,5]  
[False,True,False,True,False]
```

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

$\text{map} \dots = ?$

# Quiz: What is the Definition of map?

## Example


```
*Main> map even [1,2,3,4,5]  
[False,True,False,True,False]
```

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$
$$\text{map } f [] = []$$
$$\text{map } f (x:xs) = f x : \text{map } f xs$$

# Is this “Just Another Feature”?

# NO!!!

- Higher-order functions are the “heart of functional programming!”
- A higher-order function can do *much more* than a “first-order” one, because a part of its behavior is determined by the caller.
- We can replace *many similar* functions by *one* higher-order function, parameterised on the differences.



Avoid  
*copy-and-paste*  
programming



# Case Study: Summing a List

$$\begin{aligned} \text{sum } [] &= 0 \\ \text{sum } (x:xs) &= x + \text{sum } xs \end{aligned}$$

## General Idea

Combine the elements of a list using an operator.

## Specific to Summing

The operator is  $+$ , the base case returns 0.

# Case Study: Summing a List

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

Replace 0 and + by parameters -- + by a *function*.

```
foldr op z [] = z
foldr op z (x:xs) = x `op` foldr op z xs
```

# Case Study: Summing a List

## New Definition of sum

```
sum xs = foldr plus 0 xs
  where plus x y = x+y
```

or just...

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

Just as `fun` lets a function be used as an operator,  
so (op) lets an operator be used as a function.

# 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
```

# An Intuition About foldr

$$\begin{aligned}\text{foldr op z []} &= z \\ \text{foldr op z (x:xs)} &= x \text{ `op`} \text{foldr op z xs}\end{aligned}$$

## Example

$$\begin{aligned}\text{foldr op z (a:(b:(c:[])))} &= a \text{ `op`} \text{foldr op z (b:(c:[]))} \\ &= a \text{ `op`} (b \text{ `op`} \text{foldr op z (c:[])}) \\ &= a \text{ `op`} (b \text{ `op`} (c \text{ `op`} \text{foldr op z []})) \\ &= a \text{ `op`} (b \text{ `op`} (c \text{ `op`} z))\end{aligned}$$

The operator “:” is replaced by `op`, [] is replaced by z.

# Quiz

What is

`foldr (:) [] xs`

# Quiz

What is

`foldr (:) [] xs`

Replaces “:” by “:”, and [] by [] -- *no change!*

The result is equal to `xs`.

# Quiz

What is

`foldr (:) ys xs`



# Quiz

What is

`foldr (:) ys xs`

`foldr (:) ys (a:(b:(c:[])))`

`= a:(b:(c:ys))`

The result is `xs++ys`!

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

# Quiz

What is

`foldr snoc [] xs`

**where** `snoc y ys = ys++[y]`

# Quiz

What is

`foldr snoc [] xs`

**where** `snoc y ys = ys++[y]`

`foldr snoc [] (a:(b:(c:[])))`

`= a `snoc` (b `snoc` (c `snoc` []))`

`= (([] ++ [c]) ++ [b] ++ [a])`

The result is reverse xs!

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

# $\lambda$ -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  $\lambda$ -expression lets us define it where it is used.

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

**Just the same as**

```
unlines xss = foldr join [] xss
```

**where** join xs ys = xs ++ "\n" ++ ys

# Another Useful Pattern

Example: `takeLine "abc\ndef" = "abc"`

used to define lines.

```
takeLine [] = []
takeLine (x:xs) | x /= '\n' = x:takeLine xs
                 | otherwise = []
```

## General Idea

Take elements from a list while a condition is satisfied.

## Specific to `takeLine`

The condition is that the element is not `'\n'`.

# Generalising takeLine

```
takeLine [] = []  
takeLine (x:xs) | x/= '\n' = x : takeLine xs  
                | otherwise = []
```

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

## **New Definition**

takeLine xs = takeWhile (\x -> x/= '\n') xs

*or* takeLine xs = takeWhile (/= '\n') xs

# Notation: Sections

As a shorthand, an operator with *one* argument stands for a function of the other...

- `map (+1) [1,2,3] = [2,3,4]`
- `filter (<0) [1,-2,3] = [-2]`
- `takeWhile (0<) [1,-2,3] = [1]`

$$(a+) b = a+b$$
$$(+a) b = b+a$$

Note that expressions like `(*2+1)` are not allowed.

Write `\x -> x*2+1` instead.



# Defining lines

We use

- `takeWhile p xs` -- returns the longest *prefix* of `xs` whose elements satisfy `p`.
- `dropWhile p xs` -- returns the rest of the list.

```
lines [] = []  
lines xs = takeWhile (/='\\n') xs :  
            lines (drop 1 (dropWhile (/='\\n') xs))
```

**General idea**

Break a list into segments whose elements share some property.

**Specific to lines**

The property is: are not newlines.

# Quiz: Properties of takeWhile and dropWhile

`takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]`

Can you think of a property that connects  
takeWhile and dropWhile?

**Hint:** Think of a property that connects take and drop

Use **import**  
Text.Show.Functions

```
prop_TakeWhile_DropWhile p xs =  
  takeWhile p xs ++ dropWhile p xs == (xs :: [Int])
```

# Generalising lines

```
segments p [] = []
```

```
segments p xs = takeWhile p xs :
```

```
                segments p (drop 1 (dropWhile p xs))
```

## Example

```
segments (>=0) [1,2,3,-1,4,-2,-3,5]  
= [[1,2,3], [4], [], [5]]
```

segments is  
not a standard  
function.

```
lines xs = segments (/='\\n') xs
```

# Quiz: Comma-Separated Lists

Many Windows programs store data in files as “comma separated lists”, for example

1,2,hello,4

Define `commaSep :: String -> [String]`

so that `commaSep “1,2,hello,4” = [“1”, “2”, “hello”, “4”]`

# Quiz: Comma-Separated Lists

Many Windows programs store data in files as “comma separated lists”, for example

1,2,hello,4

Define `commaSep :: String -> [String]`

so that `commaSep “1,2,hello,4” = [“1”, “2”, “hello”, “4”]`

`commaSep xs = segments (/=',') xs`

# Defining words

We can *almost* define words using segments -- but

segments (not . isSpace) "a b" = ["a", "", "b"]

Function composition

$$(f . g) x = f (g x)$$

which is not what we want -- there should be no empty words.

words xs = filter (/="") (segments (not . isSpace) xs)

# Partial Applications

Haskell has a trick which lets us write down many functions easily. Consider this valid definition:

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

foldr was defined with  
3 arguments. 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 Applications

Any function may be called with fewer arguments than it was defined with.

The result is a *function* of the remaining arguments.

If  $f :: \text{Int} \rightarrow \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool}$

then  $f\ 3 :: \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool}$

$f\ 3\ \text{True} :: \text{Int} \rightarrow \text{Bool}$

$f\ 3\ \text{True}\ 4 :: \text{Bool}$

# Bracketing Function Calls and Types

We say            function application “brackets to the left”  
                      function types “bracket to the right”

If             $f :: \text{Int} \rightarrow (\text{Bool} \rightarrow (\text{Int} \rightarrow \text{Bool}))$   
then         $f\ 3 :: \text{Bool} \rightarrow (\text{Int} \rightarrow \text{Bool})$   
               $(f\ 3)\ \text{True} :: \text{Int} \rightarrow \text{Bool}$   
               $((f\ 3)\ \text{True})\ 4 :: \text{Bool}$

Functions really take only *one* argument, and return a function expecting more as a result.

# Designing with Higher-Order Functions

- Break the problem down into a series of small steps, each of which can be programmed using an existing higher-order function.
- Gradually “massage” the input closer to the desired output.
- Compose together all the massaging functions to get the result.

# Example: Counting Words

## Input

A string representing a text containing many words. For example

“hello clouds hello sky”

## Output

A string listing the words in order, along with how many times each word occurred.

“clouds: 1\nhello: 2\nsky: 1”

```
clouds: 1
hello: 2
sky: 1
```

# Step 1: Breaking Input into Words

“hello clouds\nhello sky”



words

[“hello”, “clouds”, “hello”, “sky”]

# Step 2: Sorting the Words

["hello", "clouds", "hello", "sky"]



sort

["clouds", "hello", "hello", "sky"]

# Digression: The groupBy Function

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

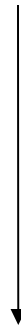
`groupBy p xs` -- breaks `xs` into segments `[x1,x2...]`, such that `p x1 xi` is `True` for each `xi` in the segment.

`groupBy (<) [3,2,4,1,5] = [[3], [2,4], [1,5]]`

`groupBy (==) "hello" = ["h", "e", "ll", "o"]`

# Step 3: Grouping Equal Words

["clouds", "hello", "hello", "sky"]



groupBy (==)

[["clouds"], ["hello", "hello"], ["sky"]]



# Step 4: Counting Each Group

```
[[“clouds”], [“hello”, “hello”], [“sky”]]
```

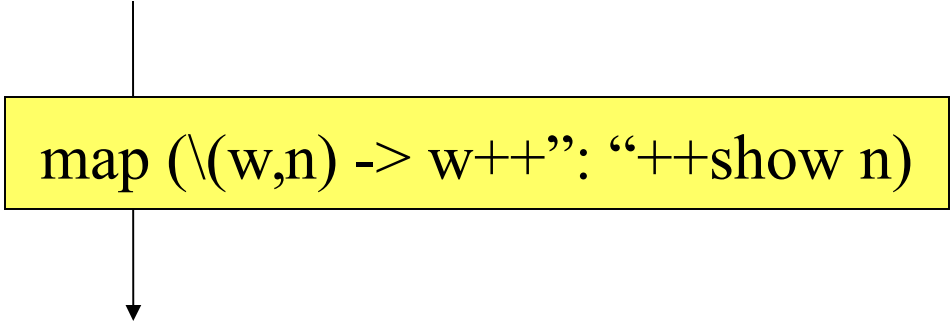


```
map (\ws -> (head ws, length ws))
```

```
[(“clouds”,1), (‘hello”, 2), (‘sky”,1)]
```

# Step 5: Formatting Each Group

[("clouds",1), ("hello", 2), ("sky",1)]



```
map (\(w,n) -> w++": "++show n)
```

[“clouds: 1”, “hello: 2”, “sky: 1”]

# Step 6: Combining the Lines

["clouds: 1", "hello: 2", "sky: 1"]



unlines

"clouds: 1\nhello: 2\nsky: 1"


```
clouds: 1
hello: 2
sky: 1
```

# The Complete Definition

```
countWords :: String -> String
```

```
countWords = unlines
```

- map (\(w,n) -> w++": "++show n)
- map (\ws -> (head ws, length ws))
- groupBy (==)
- sort
- words



very common  
coding pattern

# Quiz: A property of Map

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

Can you think of a property that merges two consecutive uses of map?

$\text{map } f (\text{map } g \text{ xs}) == ??$

```
prop_MapMap :: (Int -> Int) -> (Int -> Int) -> [Int] -> Bool
prop_MapMap f g xs =
  map f (map g xs) == map (f . g) xs
```

# The Optimized Definition

```
countWords :: String -> String
```

```
countWords = unlines
```

- `map (\ws -> head ws ++ “:” ++ show (length ws))`
- `groupBy (==)`
- `sort`
- `words`

# List Comprehensions

- List comprehensions are a different notation for **map** and **filter**
- $[ x * 2 \mid x \leftarrow xs ]$ 
  - `map (\x -> x*2) xs`
- $[ y \mid y \leftarrow ys, y \geq 3 ]$ 
  - `filter (\y -> y >= 3) ys`
- $[ z \text{ `div` } 2 \mid z \leftarrow zs, \text{ even } z ]$ 
  - `map (\z -> z `div` 2) (filter even zs)`

# List Comprehensions (2)

- More complicated list comprehensions also involve **concat**
- Example:  $[ x + y \mid x \leftarrow xs, y \leftarrow ys ]$ 
  - Quiz: How to define using map and concat?

```
concat (map (\x -> map (x+) ys) xs)
```



# concatMap

- `concat (map f xs)` is a very common expression
  - `concatMap :: (a -> [b]) -> [a] -> [b]`
- Quiz: How to define `filter` with `concatMap`?

```
filter p = concatMap (\x -> if p x then [x] else [])
```

# Where Do Higher-Order Functions Come From?

- We observe that a similar pattern recurs several times, and 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!”*

# Lessons

Avoid  
*copy-and-paste*  
programming!

- 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.
- $\lambda$ -expressions, partial applications, 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

Chapter 6 in [Learn You a Haskell](#)