Higher Order Functions

Winter 2015-2016

Table of Contents

Currying: review

- In Haskell, every function is curried by default
- We saw how currying works in the Lambda Calculus

```
bar ::
Int -> Int -> Int
bar a b = 2*a + b - 1
```

$$bar = \lambda a.\lambda b.(2\times a+b-1)$$

- Currying is the technique of writing a function of 2 or more parameters as a sequence of functions of one parameter each.
- Every function in Haskell is curried.

Function application has highest precedence

```
bar :: Int -> Int -> Int
bar a b = 2*a + b - 1
```

- `bar 4` is a legal expression, whose value is a function of `b`
- In the Lambda Calculus: $\lambda b(2 \times 4 + b - 1)$
- The expression `bar 3 + 4` causes an error
- . . . because Haskell sees + as the 2nd argument to `bar`

Some simple functions

Haskell:

```
square :: Num a => a -> a
square x = x * x

double :: Num a => a -> a
double x = x + x

jam :: [a] -> [a]
jam x = x ++ x
```

Lambda Calculus:

$$square = \lambda x.(x \times x)$$

$$double = \lambda x.(x + x)$$

$$jam = \lambda x.(x + +x)$$

The only difference between the three functions is the operator.
Goal: write a function that generalizes this kind of operation.

Generalizing

Haskell:

```
twops f x = f x x
```

Lambda Calculus:

$$twops = \lambda f.\lambda x(f\ x\ x)$$

- It's a function of 2 arguments
  1. A function f
  2. A valid input for f
- The result is f applied to x and x
- Prefix vs. infix is irrelevant here.
- The key idea is that this function "factored out" the function from three previous examples.

Example usage

```
Main> twops (+) 3
6
Main> twops (*) 3
9
Main> twops (++) [1,2,3]
 [1,2,3,1,2,3]
Main> twops (**) 3
27.0
Main> twops (&&) True
True
Main> twops (*) (twops (+) 10)
400
```

How to use twops

C/Java programmer:

```
double :: Num a => a -> a
double x = twops (+) x

square :: Num a => a -> a
square x = twops (*) x

jam :: [a] -> [a]
jam x = twops (++) x
```

Haskell programmer:

```
double :: Num a => a -> a
double = twops (+)

square :: Num a => a -> a
square = twops (*)

jam :: [a] -> [a]
jam = twops (++)
```

The key to this example is the use of currying in expressions to create functions as value.

Deducing the type signature for twops

```
twops f x = f x x
```

- LHS: twops must be a function, because it is the left-most symbol. It has 2 arguments.
- RHS: f must be a function, because it is the left-most symbol. It also has 2 arguments.
- We do not know the type of x; let's say type a
- f takes two a s, and returns something. Call it b.
- So we have deduced f :: a -> a -> b.
- (That's only part of the way finished)

```
twops f x = f x x
```

- `twops` takes two arguments, the function `f` and an `x`
- Therefore: `twops :: (a -> a -> b) -> a -> ???`
- `twops` returns whatever `f` returns, i.e., a `b`
- Therefore we can write:

```
-- twops op x
-- apply the binary operator to the value x

twops :: (a -> a -> b) -> a -> b
twops f x = f x x
```

Functions: Curying Example 3

```
-- compute both roots of quadratic equation
-- a * x * x + b * x + c = 0

quad :: Float -> Float -> Float -> (Float, Float)
quad a b c
  | a == 0      = error "not quadratic"
  | disc < 0    = error "not real"
  | otherwise   = (r1, r2)
  where disc = b*b - 4*a*c
        r1   = (-b + (sqrt disc))/(2 * a)
        r2   = (-b - (sqrt disc))/(2 * a)
```

```
Main> quad 2

<interactive>:1:1:
    No instance for (Show (Float -> Float
          -> (Float,Float))
      arising from a use of `print'
    Possible fix:
      add an instance declaration for (Show (Float
          -> Float -> (Float,Float))
    In a stmt of an interactive GHCi command: print it
```

Haskell is unable to display functions, but the functions are perfectly good values.

Functions: Currying Example 3

- `quad` evaluates to a function of three arguments
- `quad 3` evaluates to a function of two arguments
- `quad 3 7` evaluates to a function of one argument
- `quad 3 7 (−4)` evaluates to `(0.4748096,−2.808143)`

Currying with infix functions

- Suppose you needed a function to add 1 to a number.

```
incr :: Num a => a -> a
incr x = x + 1
```

- Not very exciting, and completely unnecessary!
- A function can be created on demand using currying: `(+ 1)`
- You can use `(+ 1)` anywhere you might use `incr`

More currying of infix functions

- A function to multiply by 2: `(2 *)`
- A function to check for zero: `(== 0)`
- A function to append a phrase to a string: `(++ " lol")`
- Any binary infix operator can be used this way.
- A function to put 3 on the front of a list: `(3 :)`

The Gravitational Force Example

```haskell
-- model the force of gravity between 2 masses

gravForce :: Float -> Float -> Float -> Float
gravForce mass1 radius mass2 =
  if (radius == 0) then
    0
  else
    gravConstant * mass1 * mass2 / square radius

weightOnEarth :: Float -> Float
weightOnEarth = gravForce earthMass earthRadius

sunGravity :: Float -> Float -> Float
sunGravity = gravForce sunMass
```

Functions: Currying summary

- Each function takes one argument only,
- A Haskell function produces exactly one result (the result could be a function)
- A function can be defined with a sequence of single arguments.
- Any function can be applied with a sub-sequence of those arguments in the left-to-right order.
- A function applied to a sub-sequence of arguments returns a function whose arguments are the remainder of the sequence
- The type signature of a curried function looks like a sequence of arguments, e.g.:

```
plus :: Integer -> Integer -> Integer
```

Haskell's version of the application rule: Closures

```
bar :: Int -> Int -> Int
bar a b = 2*a + b - 1
```

- bar 4 is a legal expression, whose value is a function of b
- In the lambda calculus: $\lambda b.(2 \times 4 + b - 1)$
- Re-writing is too expensive!
- Instead, create a "frame" and store $a = 4$ in the frame.
- Store the frame on the heap (not the stack).
- Keep the frame connected to the expression bar 4.
- If a's value is needed, look it up!

Closures

- A closure is a function that has captured a frame which contains a value it needs. E.g.,

```
twops f x = f x x
double = twops (+)
```

- The function twops is called, with argument (+)
- A new frame is created, to store the expression passed to twops, namely: $f = (+)$.
- The new frame is used by double whenever it is called.
- The new frame cannot be garbage collected!

Motivation

- Beginning programmers (in any language) write simple programs to manipulate simple data types, numbers, strings, arrays, etc.
- Object oriented programming makes objects the primary focus

  Objects are created, garbage collected, manipulated

  OO programmers don't often manipulate functions, though.
- Functional languages specialize in manipulating functions
- A function is just a value; in OO, you cannot easily manipulate functions. In Haskell, it is one of the main tools!
- A higher order function (HOF) takes functions as arguments or creates a function as a value
- A powerful form of abstraction allowing code reuse

Motivation (2)

- For some problems, it is good to model computation as a pipeline
- Example:
  1. start with a list of all employees
  2. collect the ones who know functional programming
  3. give them all a raise
  4. print a report about the number of HQP in the company
- In functional programming, the “pipe” is a list
- The data processing is done by functions

HOF Tools: the basics

- Function composition: build a function out of two functions
- Filtering: create a list of elements with a certain property
- Mapping: apply a function to every element in the list
- Folding: summarize a list by applying an operator to the elements

Function composition

- It is common to use the output of an expression as the input to another function, eg:

```
addFive x = x + 5
square x = x * x

example1 x = square (addFive x)
example2 x = addFive (square x)
```

- This is known as "composition"

$$h(x) = f(g(x))$$

Making composition part of the language

- Mathematicians write $h = f \circ g$, and don't have to bother mentioning the parameter.
- $f \circ g$ is a function. Its behaviour is defined as follows:

$$(f \circ g)(x) = f(g(x))$$

- Of course, f and g have to have the right input and output types.
- Haskell defines the function `(.)` which is a direct implementation of composition.

The compose function

- Simplified definition:

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

compose f g x = f (g x)
```

- Takes 2 arguments, both functions
- The output of the 2nd function is the input of the first
- The result is a function
- Exercise: Write out the lambda expression for `compose`

Haskell's built-in compose function

- Because it is so useful, Haskell defines an infix version of compose
- The operator is (.)

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

(f . g) x = f (g x)
```

Examples

```
addFive x = x + 5
square x = x * x

example1 = square . addFive
example2 = addFive . square
```

- This is very concise, and equivalent to the previous definitions.
- Keep in mind that `(.)` is just a normal function in Haskell.

Analogy to Shell Pipe operator

- Haskell's `(.)` is very much like Unix's `|`, except:
  1. `(.)` is a function (`|` is shell syntax)
  2. Data flow through `(.)` is right-to-left (`|` is left-to-right)
  3. `(.)` creates a new function (`|` only pipes output)

More examples of composition

Simple arithmetic, normal:

```
f x = 5 * x + 4
```

The same function using `(.)`

```
f = (+4) . (5*)
```

This is stylistically a bad use of `(.)`, but instructive.

Is the last character in the String a question mark?

```haskell
question :: String -> Bool

question = (=='?') . last
```

This is a good use of (.)

Convert a Circle into a Square:

```
data Shape = Circle Float | Square Float

dim :: Shape -> Float
dim ( Circle  x) = x
dim (Square x) = x

convert  :: Shape -> Shape
convert = Square . dim
```

Note the use of the constructor as a function!

Convert a Circle into a Square of same area:

```
convert2 :: Shape -> Shape
convert2 = Square . ((sqrt pi) *) . dim
```

The value of composition is in building functions “on the fly”
We will need more tools to see the full value.

Higher order functions on Lists: map

- **map** applies a given function to every element of a list
- Examples:
  - Multiply every number in a list by 3
  - Check if every letter in the list is lower case
  - Process all the data in the list
- In Haskell **map** is as important to lists as for (...) is to arrays in C, Java, etc

```
Main*> map (* 3) [1,2,3,4]
 [3,6,9,12]

Main*> map (\x->(x*x)) [1,2,3]
 [1,4,9]
```

Defining map in Haskell

- The desert island definition:

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

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

Examples of **map**

Many list processing functions can be written with **map**

```
replace :: Eq a => a -> a -> [a] -> [a]

replace this that list = map switch list
  where switch x
          | x == this   = that
          | otherwise   = x
```

```
Main> replace 1 10 [1,2,1,3,1,4]
 [10,2,10,3,10,4]

Main> replace 'a' 'A' "abracadabra"
"AbrAcAdAbrA"
```

Examples of map

Given a list of points in 2D, move them all by a given offset:

```
type Point = (Float, Float)

 translatePoint  :: Point -> Point -> Point
 translatePoint (dx,dy) (x,y) = (x+dx, y+dy)

moveAll :: Point -> [Point] -> [Point]
moveAll by = map (translatePoint by)
```

- Process all employee data in a list...
- Plot all graphs in a list...
- Process all jobs in the list...

A Meditation on map

- Advantages
  - programmer focusses on individual elements, e.g., `square`
  - **`map`** worries about the list
  - code can be very concise
- A higher-order function like **`map`** would be useful just for these reasons alone. . .
- . . . but currying, and function composition make them much more versatile

Using map to construct functions

```
squareList :: Num a => [a] -> [a]

squareList = map sq
  where sq x = x * x
```

```
Main> squareList [1,2,3,4,5]
 [1,4,9,16,25]
```

Using map with (.)

We can use function composition to build functions for map. . .
. . . without giving them names.

```
uppercase :: String -> String

uppercase = map (toEnum . up)
  where up c = fromEnum c - fromEnum 'a' + fromEnum 'A'
```

```
Main> uppercase "shouting"
"SHOUTING"
```

Map properties

- **map** has the useful property:

$$\mathrm{map}\, f\, (\mathrm{map}\, g\, l) = (\mathrm{map}\, f\, .\, \mathrm{map}\, g)\, l = \mathrm{map}\, (f\, .\, g)\, l$$

  so you can shorten nested **map** expressions using composition
- In other words:

$$(\mathrm{map}\, f)\, .\, (\mathrm{map}\, g) = \mathrm{map}\, (f\, .\, g)$$

- **map** does not care
  - how many elements are in the list
  - what kind of elements are in the list
  - what kind of function you use

Higher order functions on Lists: filter

- Another useful function on lists: **filter**
- Given a boolean function and a list, return all the elements that satisfy the function
- Which elements are greater than 5?

```
Main*> filter (>5) [3,4,5,6,7]
[6,7]
```

- Return the even numbers

```
Main*> filter even [3,4,5,6]
[4,6]
```

Examples of using filter

Return all points inside a given rectangle
(two slides)

```
type Point = (Float, Float)

-- is a point inside a rectangle?

inside :: (Ord a, Ord b) =>
          (a,b) -> (a,b) -> (a,b) -> Bool
inside (lx,ly) (rx, ry) (x, y)
  = (x > lx) && (x < rx) && (y > ly) && (y < ry)
```

```haskell
-- return points inside a rectangle

clip :: Point -> Point -> [Point] -> [Point]
clip loleft upright =
   filter (inside loleft upright)

-- return the points outside a rectangle

clipped :: Point -> Point -> [Point] -> [Point]
clipped loleft upright =
   filter (not . (inside loleft upright))
```

```
Main> clip (5,5) (10,10)   [(3,4),(5,6),(7,8),(9,10)]
 [(7.0,8.0)]

Main> clipped (5,5) (10,10)   [(3,4),(5,6),(7,8),(9,10)]
  [(3.0,4.0),(5.0,6.0),(9.0,10.0)]
```

- Return list of all employees making more than $45,000
- Return list of processes belonging to a certain user
- ...

Defining filter in Haskell

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

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

The technique of folding

- A common task is to accumulate or summarize information from a list
  - Sum the numbers in the list
    (find the total salary of a list of employees)
  - Is a property **True** of all elements in the list?
    (are all the points inside a rectangle?)
  - What's the biggest value of the list?
- You could write simple recursive functions for each...

- Sum the numbers in a list

```haskell
-- sum the elements of the list
sumList :: Num a => [a] -> a

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

$$
\begin{aligned}
\text{sumList}\,[1,2,3,4] &= 1 + (\text{sumList}\,[2,3,4]) \\
&= 1 + (2 + \text{sumList}\,[3,4])) \\
&= 1 + (2 + (3 + \text{sumList}\,[4]))) \\
&= 1 + (2 + (3 + (4 + \text{sumList}\,[]))) \\
&= 1 + (2 + (3 + (4 + 0)))
\end{aligned}
$$

- Compare the list...

$$[1,2,3,4] = 1 : 2 : 3 : 4 : []$$

- ... with the expression

$$10 = 1 + (2 + (3 + (4 + 0)))$$

- Because $(+)$ associates right, we can also write:

$$10 = 1 + 2 + 3 + 4 + 0$$

More examples of folding

- Is everything **True**?

```
allTrue :: [Bool] -> Bool

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

- (allTrue is a form of && for lists of booleans)

$$
\begin{aligned}
&\mathrm{allTrue}\,[\mathit{True}, \mathit{True}, \mathit{False}] \\
&\quad = \quad \mathit{True} \;\&\&\; (\mathrm{allTrue}\,[\mathit{True}, \mathit{False}]) \\
&\quad = \quad \mathit{True} \;\&\&\; (\mathit{True} \;\&\&\; \mathrm{allTrue}\,[\mathit{False}])) \\
&\quad = \quad \mathit{True} \;\&\&\; (\mathit{True} \;\&\&\; (\mathit{False} \;\&\&\; \mathrm{allTrue}\,[]))) \\
&\quad = \quad \mathit{True} \;\&\&\; (\mathit{True} \;\&\&\; (\mathit{False} \;\&\&\; \mathit{True}))
\end{aligned}
$$

- Are all points inside rectangle?

```
allInside  :: Point -> Point -> [Point] -> Bool

allInside  loleft  upright  []    = True
allInside  loleft  upright  (x:xs)  = y && ys
  where y  = inside  loleft  upright  x
        ys = allInside  loleft  upright  xs
```

- Note the common shape of these programs

```
<functionName> ::  [a] -> b

<functionName> [] = <base>
<functionName> (x:xs) =
   x <op> (<functionName> xs)
```

- This technique is called "folding"
  - The list elements are "folded" together using an operator
  - The empty list is the base case; its "fold" is a base value
  - Folding "replaces" the List constructor `:` with a function;
  - Folding "replaces" the empty list `[]` with a base value

- We can generalize this technique by writing a function **foldr** that abstracts the function, and the base value

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

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

- Notes:
  - First argument: a binary function returning type b
  - Second argument: type b (the same as the return type)
  - Third argument is a list of a
  - **foldr** returns values of type b

Understanding foldr

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

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

- We (generally) use **foldr** to compute a summary or transformation of a list.
- The value e is the summary for the empty list. It is provided by the programmer, because Haskell cannot guess how to summarize an empty list.
- The function f has 2 arguments
  1. The first element of the list x
  2. A summary of the tail xs of the list (computed recursively by **foldr**)

  foldr sends both of these to f

- We can define some well-known functions using **foldr** . . .

```
sumList = foldr (+) 0

allTrue = foldr (&&) True

length = foldr oneplus 0
  where oneplus x n = 1 + n

reverse = foldr rcons []
  where rcons x xs = xs ++ [x]
```

- It takes practice to use **foldr** (and other higher order functions)

Using foldr

- Trickiest part: choosing the right 2nd argument (the “base”)
  - Let f be the binary function
  - e is the “unit” value for f if
    for every y of type b: $f\ e\ y = y$
  - The 2nd argument must be e
- Examples
  - When f is $(+)$, e is 0
  - When f is $(*)$, e is 1
  - When f is (&&), e is **True**

- Can we write **map** in terms of **foldr**?
- Can we write **filter** in terms of **foldr**?

```
map f = foldr (cons . f) []
  where cons x xs = x:xs

filter p = foldr ifcons []
  where ifcons x y
          | p x       = x : y
          | otherwise = y
```

Et Cetera

- Of course, there are many more functions like this:

foldr1		
foldl	foldl'	foldl1
scanl	scanl1	
scanr	scanr1	
takeWhile	dropWile	zipWith
until		

- We will not look at all of these; you will not be examined on them
- If you really want to learn Haskell well, you will pick them up eventually.

The Student Database Example

- Suppose we had a student database for a course, including active and withdrawn students
- The database would record grades for course work, etc.
- We will compose programs for this database to do usual calculations for averages, class averages, etc.

Definition and example database

```
type Name = String
type Number = String
type Marks = [(Int,Int)]

data Student = Active    Number Name Marks
             | Withdrawn Number Name
     deriving (Show)

cmpt123 = [
       Active "123" "al"   [(10,10),(17,20),(6,10)]
      ,Active "456" "bo"   [(7,10),(9,10),(10,10)]
      ,Withdrawn "789" "cam"
      ,Active "351" "des"  [(8,10),(14,20),(10,10)]
      ,Active "963" "el"   [(4,10),(16,20),(9,10)]
      ]
```

Monolithic function to calculate average

```
-- calculate the average grade of given list of Students
classAve :: [Student] -> Int
classAve cls = cscan cls 0 0 where
  cscan [] acc n = div acc n
  cscan ((Active _ _ ls):xs) acc n
    = cscan xs ((grade ls) + acc) (n+1)
  cscan ((Withdrawn _ _):xs) acc n
    = cscan xs acc n
  grade [] = 0
  grade (x:xs) = gradeHelper (x:xs) 0 0
    where gradeHelper [] n d = div n d
          gradeHelper ((a,b):xs) n d
            = gradeHelper xs (n + (a*100)) (d+b)
```

Disadvantages of the monolithic approach

- This function makes no attempt at code re-use
- It is hard to understand and debug

Breaking the problem into steps

- Only the active students contribute towards the average
- Each student's mark must be calculated from a list of pairs
- The average is computed from a list of marks

Some accessor functions

```
getMarks :: Student -> Marks
getMarks (Active _ _ s) = s


isActive  :: Student -> Bool
isActive  (Active _ _ _) = True
isActive  _ = False
```

Calculating a final mark given some marks

```
*Main> calcSdtMark [(10,10),(17,20),(6,10)]
82
*Main> calcSdtMark [(10,10),(20,20),(10,10)]
100
*Main> calcSdtMark [(0,10),(0,20),(0,10)]
0
*Main> calcSdtMark [(8,10),(12,15),(16,20)]
80
*Main> calcSdtMark [(8,10),(12,15),(16,20),(14,37)]
60
*Main> calcSdtMark []
*** Exception: divide by zero
*Main> calcSdtMark [(5,6)]
83
```

Calculating a final mark given some marks

```
calcSdtMark :: Marks -> Int
calcSdtMark ms = div n d
  where (n,d) = collectNumDenom ms

collectNumDenom :: Marks -> (Int,Int)
collectNumDenom [] = (0,0)
collectNumDenom ((n,d):ms) = (100*n+ns,d+ds)
  where (ns,ds) = collectNumDenom ms
```

This is a nice subtask, and the recursive structure is relatively straight-forward. But it does not use HOFs!

Using foldr

Compare:

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

```
collectNumDenom [] = (0,0)
collectNumDenom ((n,d):ms) = (100*n+ns,d+ds)
  where (ns,ds) = collectNumDenom ms
```

There is some similarity in the structure. How can we rewrite using **foldr** ?

Using foldr: step 1

Make things more complicated...

```
score :: (Int,Int) -> (Int,Int) -> (Int,Int)
score (n,d) (ns,ds) = (100*n+ns, d+ds)

collectNumDenom [] = (0,0)
collectNumDenom ((n,d):ms) = score (n,d) (ns,ds)
  where (ns,ds) = collectNumDenom ms
```

Using foldr: step 2

Simplify a little

```
score :: (Int, Int) -> (Int,Int) -> (Int,Int)
score (n,d) (ns,ds) = (100*n+ns, d+ds)

collectNumDenom [] = (0,0)
collectNumDenom (m:ms) = score m (collectNumDenom ms)
```

- Replaced (n,d) with m.
- Eliminated **where** by subsitution of equal expressions

Using foldr: step 3

Compare again:

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

```
collectNumDenom [] = (0,0)
collectNumDenom (m:ms) = score m (collectNumDenom ms)
```

Now the similarity is obvious!

- Base case: 0 vs. (0,0)
- Operator: (+) vs. score

Using foldr: step 4

Rewrite using foldr...

```
score :: (Int,Int) -> (Int,Int) -> (Int,Int)
score (n,d) (ns,ds) = (100*n+ns, d+ds)

collectNumDenom ms = foldr score (0,0) ms
```

Using foldr: step 5

Exploit currying, and tidy up!

```
collectNumDenom = foldr score (0,0)
  where score (n,d) (ns,ds) = (100*n+ns, d+ds)
```

RE: Calculating a final mark given some marks

```
calcSdtMark :: Marks -> Int
calcSdtMark ms = div n d
  where (n,d) = collectNumDenom ms

collectNumDenom :: Marks -> (Int,Int)
collectNumDenom = foldr score (0,0)
  where score (n,d) (ns,ds) = (100*n+ns, d+ds)
```

Can we simplify calcSdtMark? Yes, a little!

Simplifying CalcStudentMark

Simplify by introducing an auxiliary function:

```
divpair  :: (Int,Int) -> Int
divpair  (n,d) = div n d

calcSdtMark :: Marks -> Int
calcSdtMark ms = divpair (n,d)
  where (n,d) = collectNumDenom ms
```

Simplifying CalcStudentMark

By replacing equal quantities

```
divpair  :: (Int,Int) -> Int
divpair  (n,d) = div n d

calcSdtMark :: Marks -> Int
calcSdtMark ms = divpair (collectNumDenom ms)
```

I removed the where clause, by moving the expression!

Simplifying CalcStudentMark

By recognizing function composition.

```
divpair  :: (Int,Int) -> Int
divpair  (n,d) = div n d

calcSdtMark :: Marks -> Int
calcSdtMark ms = (divpair . collectNumDenom) ms
```

You have to know (.) very well!

Simplifying CalcStudentMark

Exploiting currying, and tidying up:

```
calcSdtMark :: Marks -> Int
calcSdtMark = (divpair . collectNumDenom)
  where divpair (n,d) = div n d
```

I dropped the argument, and added a where clause.

So far, so good

```
calcSdtMark :: Marks -> Int
calcSdtMark = divpair . collectNumDenom
  where divpair (n,d) = div n d

collectNumDenom :: Marks -> (Int,Int)
collectNumDenom = foldr score (0,0)
  where score (n,d) (ns,ds) = (100*n+ns, d+ds)
```

But we can simplify more! Replace collectNumDenom with its equivalent expression

Final Version

```
calcSdtMark :: Marks -> Int
calcSdtMark = divPair . (foldr score (0,0))
  where divPair (a,b)       = div a b
        score (x,y) (u,v)   = (100*x+u,y+v)
```

Don't let the novelty of the tool influence your assessment.

Processing the class list

- Given a list of Students
- Take the active students
- Extract their mark lists
- Calculate their mark in the class
- Put all the marks in a list

Processing the class list

Examples:

```
*Main> collectMarks [Active "123" "al"  [(10,10),(17,20),(6,10)]]
[82]
*Main> collectMarks [Withdrawn "789" "cam"]
[]
*Main> collectMarks cmpt123
[82,86,80,72]
```

Processing the class list

Collect the final marks for active students into a list.

```
collectMarks :: [Student] -> [Int]
collectMarks [] = []
collectMarks ((Active _ _ g):ss)
  = calcSdtMark g : collectMarks ss
collectMarks (_:ss) = collectMarks ss
```

Notice:

- We are applying calcSdtMark to each list of marks (map)
- We are ignoring all but the Active students (filter)

Processing the class list

Rewriting by dividing the work into 3 steps:

```
collectMarks students
  = processMarks (getMarksList (onlyActive students))
  where
    onlyActive [] = []
    onlyActive (s:ss) = if isActive s
                          then s:onlyActive ss
                          else onlyActive ss
    getMarksList [] = []
    getMarksList (s:ss) = getMarks s:getMarksList ss
    processMarks [] = []
    processMarks (s:ss) = calcSdtMark s:processMarks ss
```

Processing the class list

Rewriting by using **map** and **filter**

```
collectMarks  students
  = processMarks (getMarksLists (onlyActive  students))
  where onlyActive ss     = filter  isActive  ss
        getMarksList  ss  = map getMarks ss
        processMarks ss   = map calcSdtMark ss
```

Processing the class list

Rewriting by exploiting currying:

```
collectMarks students
  = processMarks (getMarksLists (onlyActive students))
  where onlyActive    = filter isActive
        getMarksList  = map getMarks
        processMarks  = map calcSdtMark
```

Processing the class list

Rewriting by recognizing function composition:

```
collectMarks students
  = (processMarks . getMarksList . onlyActive) students
  where onlyActive    = filter isActive
        getMarksList  = map getMarks
        processMarks  = map calcSdtMark
```

Processing the class list

Rewriting by exploiting currying:

```
collectMarks = processMarks . getMarksList . onlyActive
  where onlyActive   = filter isActive
        getMarksList = map getMarks
        processMarks = map calcSdtMark
```

Processing the class list

Rewriting by replacing equal expressions:

collectMarks
  = (**map** calcSdtMark) . (**map** getMarks) . (**filter** isActive)

Processing the class list

Applying the law of map: $map\ f . map\ g = map\ (f . g)$

```
collectMarks
  = map (calcSdtMark . getMarks) . ( filter  isActive )
```

Calculating an average

- Given a list of integers.
- Calculate an average.
- In this example, the numbers are integer
- Assignment 6: You implement a version of this.
- For these slides, we assume a version of average

```
average :: [Int] -> Int
```

Putting it all together

```
classAverage :: [Student] -> Int
classAverage ss = average (collectMarks ss)
```

Putting it all together

Recognizing function composition, and exploiting currying:

```
classAverage  ::  [Student] -> Int
classAverage  = average .  collectMarks

collectMarks  ::  [Student] -> [Int]
collectMarks
  = map (calcSdtMark . getMarks) . ( filter  isActive )
```

Putting it all together

Replacing equal expressions...

```
classAverage
  = average . map (calcSdtMark . getMarks) . ( filter   isActive )
```

Final version, showing everything

```
classAverage
  = average . map (calcSdtMark . getMarks) . ( filter  isActive )

calcSdtMark = divPair . (foldr score (0,0))
  where divPair (a,b)        = div a b
        score (x,y) (u,v)    = (100*x+u,y+v)

getMarks (Active _ _ s) = s

isActive (Active _ _ _) = True
isActive _ = False
```

. . . except average, which is 1 more line of Haskell.

Review: Monolithic function to calculate average

```
-- calculate the average grade of given list of Students
classAve :: [Student] -> Int
classAve cls = cscan cls 0 0 where
  cscan [] acc n = div acc n
  cscan ((Active _ _ ls):xs) acc n
    = cscan xs ((grade ls) + acc) (n+1)
  cscan ((Withdrawn _ _):xs) acc n
    = cscan xs acc n
  grade [] = 0
  grade (x:xs) = gradeHelper (x:xs) 0 0
    where gradeHelper [] n d = div n d
          gradeHelper ((a,b):xs) n d
            = gradeHelper xs (n + (a*100)) (d+b)
```

Summary

- This example worked from novice code, to HOFs.
- With practice, you will begin to start with HOFs, especially basic filters, and maps.
- Each step in the development of this code was a derivation. We applied mathematical theorems to transform the code.
- The new version can be tested in pieces – easier to debug!
- HOFs are code reuse. We designed some special purpose functions used by **map** and **foldr**