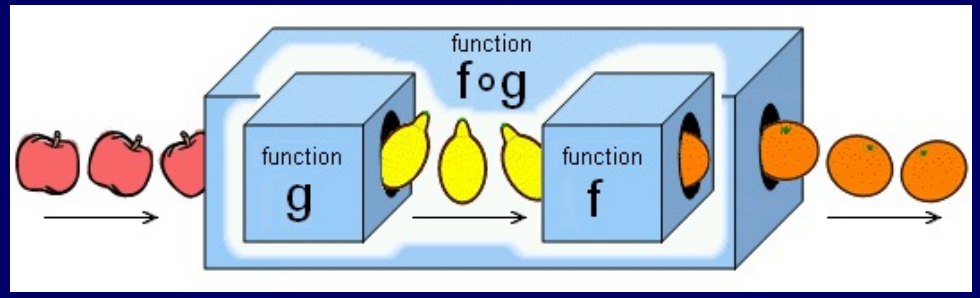
PROGRAMMING IN HASKELL

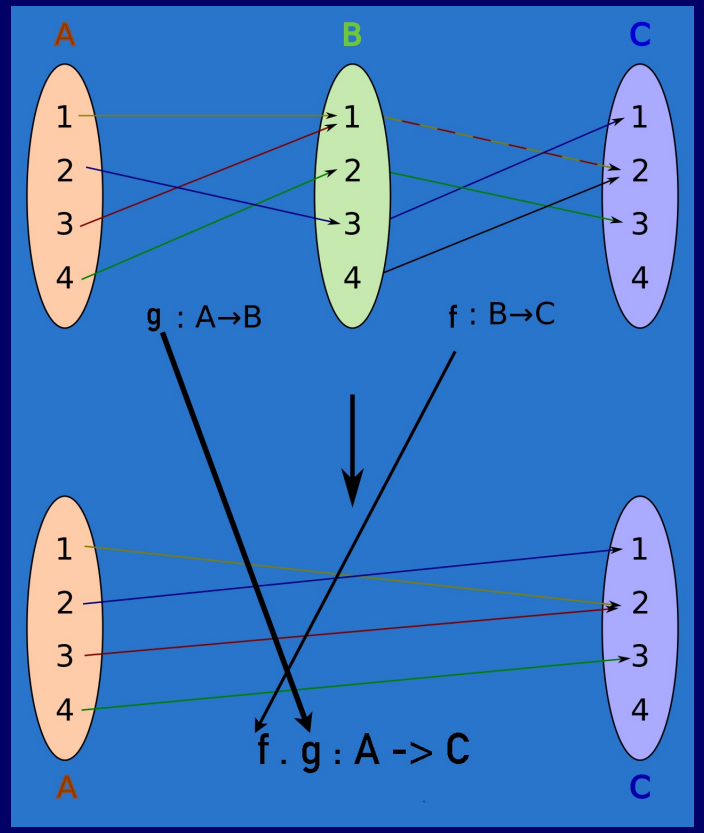
Chapter 8.3
Function Composition

Function Composition

We sometimes use one function after another and we can see these functions as one, composed together:

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





Function Composition

Call g with some value, call f with the result

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

Input parameter of f **must** be the same as the return type of g

Function Composition – why

Often convenient to create functions on the fly
Could use lambda, but composition may be more concise

```
*Main> map (\x -> negate (abs x)) [5,-3, -2, 7]
```

[-5,-3,-2,-7]

```
*Main> map (negate . abs) [5, -3, -2, 7]
```

[-5,-3,-2,-7]

Function Composition – why

```
*Main> map (\xs -> negate (sum (tail xs)))
[[1..5],[3..6],[1..7]]
    [-14,-15,-27]
*Main> map (negate . sum . tail)
[[1..5],[3..6],[1..7]]
    [-14,-15,-27]
```

Function Composition with multiple parameters

If a function takes multiple parameters, we must partially apply

```
*Main> sum (replicate 5 (max 6 9))
45
*Main> (sum . replicate 5) (max 6 9)
45
*Main> sum . replicate 5 $ max 6 9
45
```

The process

To rewrite a function with lots of parentheses using function composition

- first write out the innermost function and its parameters
- then put a $ before it
- compose all prior functions by omitting their last parameter (but not other parameters) and putting . between them

The process

```
*Main> replicate 2 (product (map(*3) (zipWith max [1,2] [4,5])))
```

[180,180]

```
*Main> replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

[180,180]

Aside on zipWith

zipWith takes a function and two lists as parameters and then joins the two lists by applying the function between corresponding elements. Here's how we'll implement it*:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

*Note we use zipWith' so that you can check this and not cause a conflict with Prelude's version

Aside on zipWith

```
ghci> zipWith (+) [4,2,5,6] [2,6,2,3]
        = [6,8,7,9]

ghci> zipWith max [6,3,2,1] [7,3,1,5]
        = [7,3,2,5]

ghci> zipWith (++) ["foo ", "bar ", "buzz "]
                        ["fighters", "hoppers", "aldrin"]
        = ["foo fighters", "bar hoppers"," buzz  aldrin"]
```

Aside on zipWith

```
ghci> zipWith (*)    (replicate 5 2) [1..]
                  [2,4,6,8,10]

ghci> zipWith (zipWith (*))
                           [[1,2,3],[3,5,6],[2,3,4]]
                           [[3,2,2],[3,4,5],[5,4,3]]

                  [[3,4,6],[9,20,30],[10,12,12]]
```

Aside on zipWith

Note that you can use function application as the function

```
zipWith ($) funcList valueList

zipWith ($) [(+ 5),(* 3)] [1,5]
        = [6,15]
```

Eta conversion with Funtion Composition

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

can be rewritten as

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

```
myfunc :: a -> b
myfunc x = (f . g . h . i . j . k )   x
```

can be rewritten as

```
myfunc :: a -> b
myfunc = (f . g . h . i . j . k )
```

Eta Conversion

```
answer :: [Int] -> Int
answer xs = sum (map cube (filter by7 xs))

cube :: Int -> Int
cube x = x * x * x

by7 :: Int -> Bool
by7 x = x `mod` 7 == 0
```

can be rewritten using the eta reduction

More on Composition Operator

```
answer :: [Int] -> Int
answer xs = sum . map cube . filter by7 xs
```

Object-level definition

can be rewritten , by removing xs when it is the rightmost term on each side of =

```
answer :: [Int] -> Int
answer = sum . map cube . filter by7
```

Function-level definition

More on Composition Operator

```
fun xs = (filter odd . map square) xs
```

similarly can be rewritten

```
fun = filter odd . map square
```

Eta abstraction

Eta reduction


ANY
QUESTIONS?