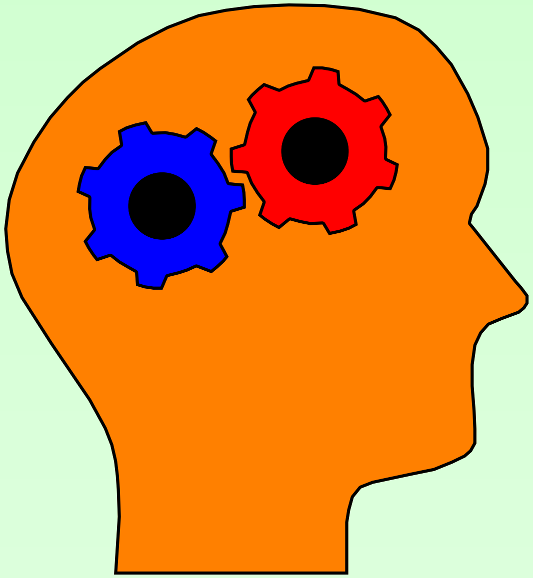
Lecture 4 : **Higher-Order Functions**



"Programming with First-Class Functions"

Lecturer : <u>Chin</u> Wei Ngan

Email : chinwn@comp.nus.edu.sg

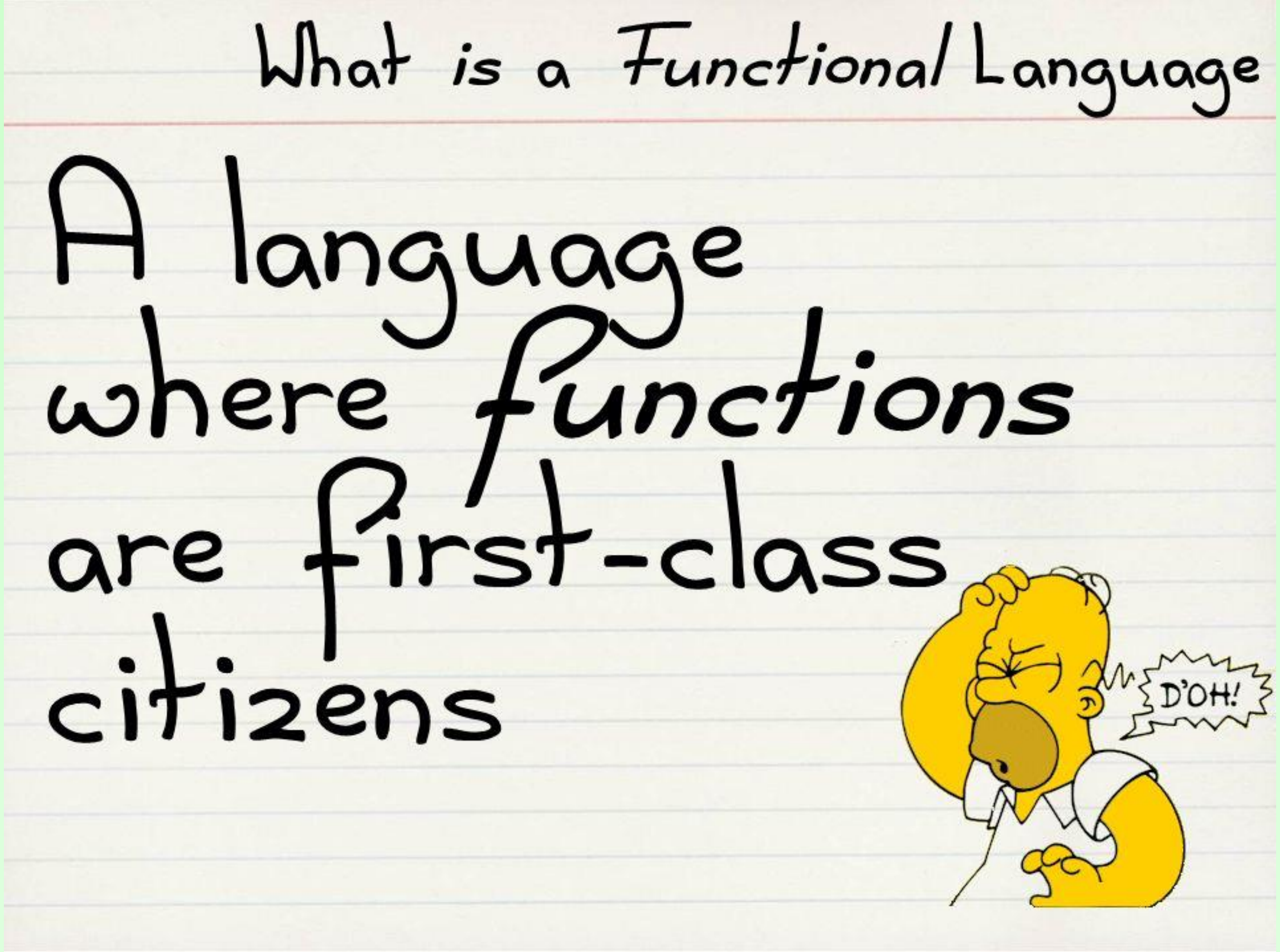
Office : COM2 4-32

Topics

- First-Class Functions

- Higher-Order Functions
    - Genericity/Parameterization
    - Fold (left and right) & Foldable
    - Map & Functor/Applicative
    - Composition
    - Pipeline
    - Application

Essence of FP



Higher-Order Functions

- Like data structures, functions should be first-class:
    - It has a value and type
    - It can be passed as arguments
    - It can be returned as function result.
    - It can be constructed at run-time
    - It can be stored inside data structures

- Why are higher-order functions useful?
    - Can support code-reuse
    - Can support laziness
    - Can support data abstraction (see OO later)
    - Can support design patterns

Functions that Returns Functions

- Function results:

```
let add x = \ y -> x+y
```

- Equivalent to curried function:

```
let add x y = x+y
```

- Can also return different functions :

```
let add_mag x =
      if x>=0 then \ y -> x + y
      else \ y -> -x + y
```

Curried vs Uncurried Functions

- Type of Curried Function:

```
a -> b -> c
```

- Type of Uncurried (or Tupled) Function:

```
(a,b) -> c
```

- These two functions are isomorphic and are inter-convertible using:

```
  curry   :: ((a,b)->c) -> (a -> b -> c)
 uncurry :: (a -> b -> c) -> ((a,b)->c)
```

- Exercise : How would you implement above?

Functions that Returns Functions

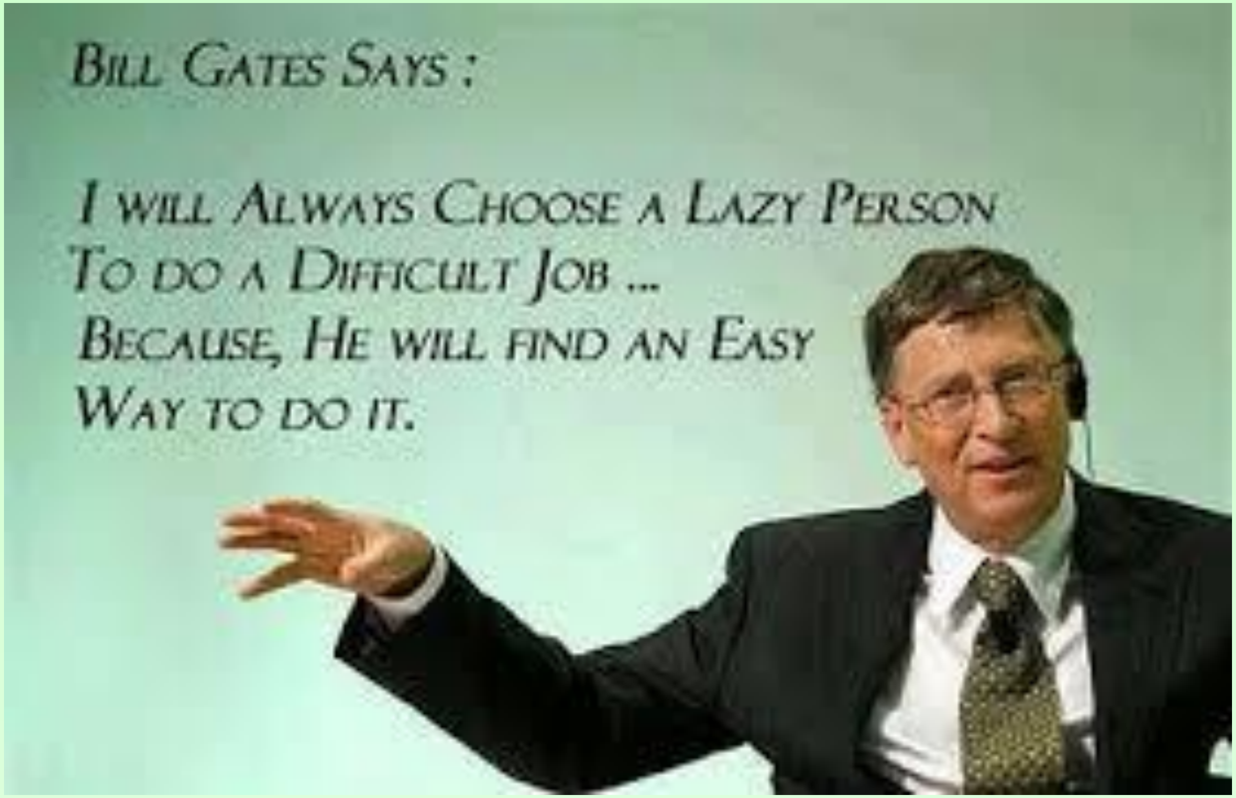
- Examples of Usages:

```
let add x = \ y -> x+y
let inc = add 1
let inc10 = add 10
let dec = add (-1)
let two = inc (1::Int)
```

- An Evaluation:

```
two
  → inc 1
  → (add 1) 1
  → (\ y -> 1+y ) 1
  → 1+1
  → 2
```

Is Laziness Good?



Lazy Evaluation via Functions

- Any given expression `e` can be abstracted into a function `(\ () -> e)` .

- This is called a *closure* which provides a way to define an expression without evaluating it.

- To evaluate the expression, we simply apply it to `()`, as follows:

```
(\ () -> e) () ==> e
```

Evaluation of expression is delayed to application.
Update of *closure* is supported to reuse result of evaluation.
If function is not applied, the expression is not evaluated.

Lazy Evaluation

- This is the default evaluation for Haskell.
- It applies to both function calls and also let construct.
- Lazy evaluation allows us to handle non-terminating code, such as `bot` below, as long as its computation is not evaluated (or required) by its context

```
let bot = bot in ....

f(…,bot,…)
```

Infinite Data Structures

- Circular structures are more space efficient :

```
ones                = 1 : ones
```

list comprehension

- Another example of circularity is.

```
fib      = 1 : 1 : [a+b | (a,b) <- zip fib (tail fib)]

zip (x:xs) (y:ys)= (x,y) : (zip xs ys)
zip xs ys         = []
```

- This circular fibonacci function can be computed very efficiently.
- Question : Is it space efficient?

Strict Data Constructors

- Strict evaluation is the converse of lazy evaluation.
- In Haskell, if strict evaluation is needed, we can mark it with `!`. This can save on memory for building closures and thus minimise memory leaks.
- An example:

```
data RealFloat a => Complex a = !a :+ !a
```

- With this, `1 :+ bot` is then equivalent to just `bot`

Strict Evaluation

- To evaluate `e1` strictly, Haskell allows:

    - `e1 ‘seq’ e2`
    - `case e1 of …`

- GHC extension also allows :

    `f $! e`
    where e is strictly evaluated

    `let f x !y !z = …`
    where y, z are strictly evaluated

- Can you correctly implement `($!)` ?

Genericity/Parameterization

- To make a function generic is to let any specific entity (i.e. operation or value) in the function body become an argument

- This parameter abstraction can help us obtain more generic program code.

Two Similar Examples

- Function to sum a list of numbers.

```
let sum xs =
      case xs of
         [] -> 0
         y:ys -> y+(sum ys)
```

- Function to multiply a list of numbers.

```
let prod xs =
      case xs of
       [] -> 1
       y:ys -> y*(prod ys)
```

- Examples :

```
 sum [1,2,3,4]  → 1+2+3+4+0
prod [1,2,3,4]  → 1*2*3*4*1
```

Genericity

- Replace each constant (that differs) by a parameter.
- Replace each function (that differs) by a parameter.

```
let sum xs =
     case xs of
      [] -> 0
      y:ys -> y+(sum ys)
```

- Generalize to :

```
let foldr xs =
     case xs of
      [] -> z
      y:ys -> f y (foldr ys)
```

Generic Fold Method

- Generalization of sum and prod gives fold.

```
let foldr f z xs =
     let aux xs =
          case xs of
            [] -> z
            y:ys -> f y (aux ys)
      in aux xs
```

- Usages :

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

let prod xs = foldr (*) 1 xs
```

Higher-Order Types

- Type of sum/prod.

```
let sum xs = …
  Type of sum :: Nat a => [a] -> a
let rec prod xs = …
  Type of prod :: Nat a => [a] -> a
```

- Type of fold :

```
let foldr f z xs = …
Type of foldr ::
  (a -> z -> z) -> z -> [a] -> z
```

Example Execution

```
foldr f z xs =
  let aux xs =
    case xs of
    | [] -> z
    | y:ys -> f y (aux ys)
  in aux xs
```

- Example of sum:

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

- Example of prod:

```
prod [1,2,3]
→ foldr (*) 1 [1,2,3]
→ aux [1,2,3]
→  * 1 (aux [2,3])
→  * 1 (* 2 (aux [3]))
→  * 1 (* 2 (* 3 aux []))
→  * 1 (* 2 (* 3 1))
```

Right Fold Method

```
foldr f z xs =
  let aux xs =
    case xs of
      [] -> z
      y:ys -> f y (aux ys)
   in aux xs
```

- Actually foldr denotes fold_right:

```
foldr f z [x1,x2,x3,x4]

→  f x1 (f x2 (f x3 (f x4 z)))
```

right is here!

Left Fold Method

- An example of foldl method which folds leftwards:

```
foldl f z [x1,x2,x3,x4]

→   f (f (f (f z x1) x2) x3) x4
```

left is here!

Fold Left Method

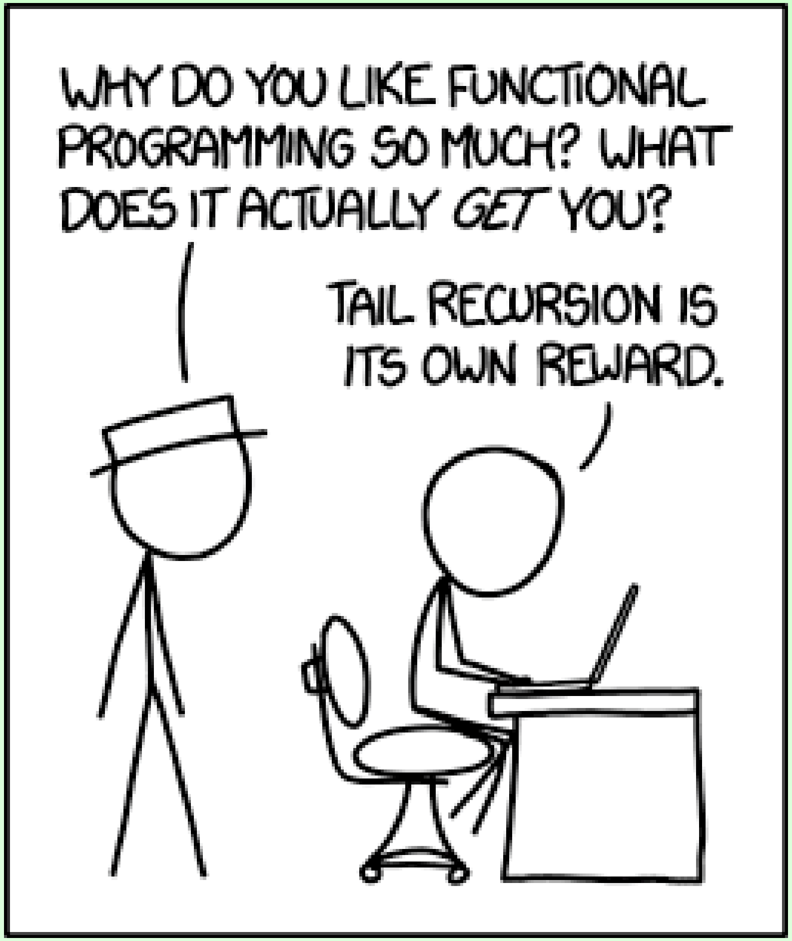
- Implementation of `foldl` method:

```
let foldl f z xs =
  let aux acc xs =
    case xs with
    | [] -> acc
    | y:ys -> aux (f acc y) ys
  in aux z xs
```

Key property : *tail-recursive*!

```
Type of foldl ::
   (z -> a -> z) -> z -> [a] -> z
```

Fold Left is Tail-Recursive



Example Execution

- Using *right fold* :

```
let foldr f z xs =
  let rec aux xs =
    match xs with
    | [] -> z
    | y::ys -> f y (aux ys)
  in aux xs
```

```
sum [1,2,3]
→ foldr (+) 0 [1,2,3]
→ aux [1,2,3]
→ + 1 (aux [2,3])
→ + 1 (+ 2 (aux [3]))
→ + 1 (+ 2 (+ 3 aux []))
→ + 1 (+ 2 (+ 3 0))
→ 6
```

- Using *left fold* :

```
let foldl f z xs =
  let rec aux acc xs =
    match xs with
    | [] -> acc
    | y::ys -> aux (f y acc) ys
  in aux z xs
```

```
sum [1,2,3]
→ foldl (+) 0 [1,2,3]
→ aux 0 [1,2,3]
→ aux (0+1) [2,3]
→ aux 1 [2,3]
→ aux 3 [3]
→ aux 6 []
→ 6
```

Fold Left or Fold Right?

- Can be transformed to each other when the reduction f operator is *associative* :

```
f a (f b c) = f (f a b) c
```

- Typically, z is the unit of f:

```
f x z = f z x
      = x
```

- In terms of performance, foldl is typically more efficient due to constant stack space. But not always!

Flattening a List of Lists

- An example: `concat [[1,2],[],[3]]` ➔ `[1,2,3]`

- Implement in terms of foldr.

```
let foldr f z xs =
  let aux xs =
    case xs of
     [] -> z
     y:ys -> f y (aux ys)
  in aux xs
```

```
concat xss
 = foldr (++) [] xss
```

- Example execution:

```
concat [[1,2],[],[3]]
  → foldr (++) [] [[1,2],[],[3]]
  → aux [[1,2],[],[3]]
  → [1,2] ++ (aux [[],[3]])
  → [1,2] ++ ([] ++ (aux [[3]]))
  → [1,2] ++ ([] ++ ([3] ++ (aux [])))
  → [1,2] ++ ([] ++ ([3] ++ []))
  → [1,2,3]
```

Flattening a List of Lists

- Implement in terms of fold_left.

```
concat xss
  = foldl (++) [] xss
```

```
let foldl f z xs =
  let aux acc xs =
    match xs with
     [] -> acc
     y:ys -> aux (f y acc) ys
  in aux z xs
```

- Example execution:

```
concat [[1,2],[],[3]]
  → foldl (++) [] [[1,2],[],[3]]
  → aux [] [[1,2],[],[3]]
  → aux ([] ++ [1,2]) [[],[3]]
  → aux (([] ++ [1,2]) ++ []) [[3]]
  → aux ((([] ++ [1,2]) ++ []) ++ [3]) []
  → ((([] ++ [1,2]) ++ []) ++ [3])
```

Fold Left versus Fold Right?

- Essential difference:

```
x1 ++ (x2 ++ (x3 ++ (x4 ++ [])))
versus
((([] ++ x1) ++ x2) ++ x3) ++ x4
```

- Which is better? Assume each of x1..x4 is 10 elements

```
x1 ++ (x2 ++ (x3 ++ (x4 ++ [])))
```

takes 10+10+10+10 steps

```
([] ++ x1) ++ x2) ++ x3) ++ x4
```

takes 0+10+20+30 steps

Foldable in Haskell

- Folding over List :

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

- Folding over Tree :

```
foldr :: (a->b->b) -> b > Tree a -> b
```

- Generic Folding :

```
foldr :: Foldable t => (a->b->b) -> b > t a -> b

    class Foldable (t :: * -> *) where
      ...
      foldr :: (a -> b -> b) -> b -> t a -> b
      ...
```

Foldable Type Class

```
class Foldable (t :: * -> *) where
      :
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldl :: (b -> a -> b) -> b -> t a -> b
  foldr1 :: (a -> a -> a) -> t a -> a
  foldl1 :: (a -> a -> a) -> t a -> a
  null :: t a -> Bool
  length :: t a -> Int
  elem :: Eq a => a -> t a -> Bool
  maximum :: Ord a => t a -> a
  minimum :: Ord a => t a -> a
  sum :: Num a => t a -> a
  product :: Num a => t a -> a
```

Examples of Foldable

```
instance Foldable [] -- Defined in 'Data.Foldable'
instance Foldable Maybe -- Defined in 'Data.Foldable'
instance Foldable (Either a) -- Defined in 'Data.Foldable'
instance Foldable ((,) a) -- Defined in 'Data.Foldable'
```

- Guess of output of these executions:

```
let foo = foldr (\x y -> x+y) 0
foo [1..4]
foo (Just 10)
foo Nothing
```

List Mapping

- Transform one list into another :

```
map f xs =
  let aux xs =
    case xs of
     [] -> []
     y:ys -> (f y):(aux ys)
   in aux xs
```

- Usages :

```
let double xs = map (fun x -> 2*x) xs

let is_pos xs = map (fun x -> x>0) xs
```

Type of Map

- Map function :

```
map f xs =
  let aux xs =
    case xs of
     [] -> []
     y:ys -> (f y):(aux ys)
   in aux xs
```

- Type of map :

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

- Note that result type of list is changed by function parameter of type `(a -> b)`.

Example Execution

```
map f xs =
  let aux xs =
        case xs of
         [] -> []
         y:ys -> (f y):(aux ys)
   in aux xs
```

Example :

```
double [1,2]
→ map (fun x -> 2*x) [1,2]
→ aux [1,2]
→ ((fun x -> 2*x) 1) : aux [2]
→ ((fun x -> 2*x) 1) : (fun x -> 2*x) 2) : aux []
→ 2 : 4 : []
→ [2,4]
```

Example Execution

```
map f xs =
  let aux xs =
           case xs of
            [] -> []
            y:ys -> (f y):(aux ys)
   in aux xs
```

Example :

```
is_pos [1,-2]
→ map (fun x -> x>0) [1,-2]
→ aux [1,-2]
→ ((fun x -> x>0) 1) : aux [-2]
→ ((fun x -> x>0) 1) : (fun x -> x>0) -2) : aux []
→ True : False : []
→ [True,False]
```

Functor in Haskell

- Mapping over List :

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

- Mapping over Tree :

```
map :: (a->b) -> Tree a -> Tree b
```

- Generic Mapping :

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
```

- Can you implement `(<$)` in terms of `fmap` :

Examples of Functors

```
instance Functor (Either a) -- Defined in 'Data.Either'
instance Functor [] -- Defined in 'GHC.Base'
instance Functor Maybe -- Defined in 'GHC.Base'
instance Functor IO -- Defined in 'GHC.Base'
instance Functor ((->) r) -- Defined in 'GHC.Base'
instance Functor ((,) a) -- Defined in 'GHC.Base'
```

- Guess of output of these executions:

```
let foo = fmap (\x -> x+1)
foo [1..10]
foo (Just 10)
foo Nothing
foo (\x -> x*2) 3
```

Composition

- We can write a general compose operator:

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

- Type of compose :

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

- Similar to Unix pipe :

```
cat x | f | g
```

Except Unix pipe was for a text file, and presented in infix notation.

Infix Version of Composition

- In Haskell, we use an infix version of `compose`

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

- Type of compose :

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

- How is compose related to `fmap` ?

```
fmap :: Functor f => (a -> b) -> f a -> f b

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

Unix Pipe in Haskell

- Declare an infix pipe operator:

```
let (|>) :: a -> (a->b) -> b
     a |> f =  f x
```

- Example of use:

```
x
|> f
|> g
```

Equivalent to:

```
(x |> f) |> g = g (f x)
```

An Example of Pipe

- Let us first declare:

```
let double xs = map (fun x -> 2*x) xs

let sum xs = foldl (+) 0 xs
```

- Example of use:

```
[1,2,3]
|> double
|> double
|> sum
```

Weak Precedence Apply Operator

- Another infix apply operator:

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

- `$` is essentially function apply but with very weak precedence:

- Example of use:

```
inc $ x*2
= inc (x*2)
```

- Without `$`, the default application gives:

```
inc x*2
= (inc x)*2
```