COURSE NAME: CS558

COURSE DATE: 8/30/18

1. Lecture
   1. Professor did review from last time on Caps and filter list:

filter abstracts the operation

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

filter p [ ] = [ ]

filter p (x: xs) = if px then x: (filter p xs)

else filter p xs

* + 1. This is a higher order function. It takes a function and essentially produces another function.
       1. We can use this filter for our Caps:

onlyCaps xs = filter isCap xs

Or noCaps, using anonymous function:

noCaps xs = filter (ln -> not (isCap xs))

The anonymous function is in ( parenthesis ).

* + 1. Last time, we also looked at map.
       1. It applies a function to each element in a list.
          1. Type signature for map:

map .. (a -> b) -> [a] -> [b]

* + - * 1. Example:

map (ln -> n +1) [2, 4, 6]

= ((ln -> n+1) 2) : (map (ln -> n+1) [4, 6])

* + - 1. Take a list of strings (list of lists!):

[“Hello”, “World!”] : : [String ] = [ [ Char] ]

[ [ ‘H’, ‘e’, ‘l’, ‘l’, ‘o’,], [‘w’,’o’,’r’,’l’,’d’,’!’,]]

Use isCap to get just the caps from each string in the list.

* + 1. Folding 🡪 there’s both a left and right side!

foldr .. (a -> b -> b) -> b -> [a] -> b

foldr f v [ ] = v

foldr f v (x: xs) = f x (foldr f v xs)

foldr f v (x0 : (x1 : (x2 : [ ] )))

F x0 (f x1 (f x2 v ) ) )

( ( : ) x0 ( ( : ) x 1 ( ( .) x2 [ ] ) ) )

Then sum the lists:

sum .. [ Int ] 🡪 Int

sum xs = foldr (+) 0 xs

sum = foldr (+) 0

sum [1, 2] = foldr (+) 0 [1, 2]

= 1 + (foldr (+) 0 [2] )

= 1 + (2+ (foldr (+) 0 [ ] ]))

= 1 + (2 + )) = 3

foldr (+) 0 ( ( : ) 1 ( ( : ) 2 [ 3] ))

You can also use product:

product =- foldr (\*) 1

* + 1. Or implement map using foldr. Professor showed an example using both map and filter with foldr.
       1. The trick is to put what filter does into the function you’re folding:

filter p xs = folder ….

* + 1. There is also foldl

foldl ….. (b -> a -> b) -> b ->[a] -> b

foldl f v [ ] = v

foldl f v (x: xs) = foldl f (f v x ) xs

* + - 1. So v is used to store.
      2. An example using a nonempty list:

sum = foldl (+) 0

sum [1, 2] = foldl (+) 0 [1, 2]

= foldl (+) (0 + 1) [2]

= foldl (+1) (( 0 + 1) + 2) [ ]

= (( ) + 1 ) + 2) = 3

* + - * 1. With the first foldr, we pushed in foldr every time we did a conversion and then it disappeared. With foldl, it is always in the top left of the expression.
        2. These operations are associative.

Some operations are not associative. Subtraction is not associative. We would get a different answer.

* + - 1. Properties

Map : (a -> b) -> [a] -> [b]

* + - * 1. Desirable properties:

map id = id (when id x = x)

map (f g) = (map f), (map g) where (f .g) x = f(g x)

How do we proves these properties hold for all lists?

You need to show that they would get the same results for all input lists.

Prove by induction (because you’re proving for all lists).

First, prove the base case.

Prove the inductive case.

The inductive case involves adding an element to the list (not just 1 + number).

You might add many different elements to the list.

You have to quantify the list! 🡪 The principle of structural induction on lists.

If you want to prove for all lists, you need to prove base case and prove inductive case.

##end notes##

##end notes##