COURSE NAME: CS558

DATE: 08/28/18

1. LECTURE
   1. Last time, we talked about what kinds of lists we can make.
      1. Empty list contains no elements [ ]
      2. Non-empty list is constructed

Example: x : 25, where x is type a and 25 is type [a] and the whole thing is type [a]

These are the same:

[1,2,3] : : 1: (2: ( 3: [ ] ))

This takes an element of any type and returns int:

Length [a] 🡪 Int

Length [ ] = 0

Length (x, xs) = 1 + length (xs)

* + 1. The take function returns the first elements of the list:

take Int -> [a] -> [a]

take\_[ ] = [ ]

take 0 \_ : [ ]

* + - 1. The underscore is a wildcard and an be used for an argument, but it can’t appear on the right side!
      2. Drop function removes the first part of the list

drop Int -> [a] -> [a]

drop\_ [ ] = [ ]

drop 0 xs = xs

drop n (x -> xs) = drop (n-1) xs

* + - 1. Returning a single element from list:

(!!) [a] -> Int -> a

[ ] !! n = empty list

(x xs) !! 0 = x

(x xs) !! n = xs !! (n-!)

* + 1. It can be helpful to determine if a particular element is present:

elem a -> [a] -> Bool

x ‘elem’ [ ] = False

x ‘elem’ (y: ys) = if x - - y then True else x ‘elem’ ys

* + 1. Strings are lists of characters

string \_\_\_\_\_\_\_\_\_\_\_\_\_\_ [ Char]

“Hello” = = (‘H’ ( ‘e’ ( ‘l’ ( ‘l’ ( ‘o’ : [ ] ) ) ) ) )

‘c’ ‘ elem’ “Hello” = if ‘e’ \_ \_ “H” then True else

‘e’ ‘elem’ “Hello”

= if ‘e’ == ‘e’ then True else ..

= True

c is Char. “Hello” is the string.

* + - 1. Is there a way to abstract an operation into higher level functions to define multiple strings?
      2. Yes 🡪 list functionals! They take functions in arguments and create lists.

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

The filter is a higher order function and takes a function as an argument.

Think of the function as a predicate on ‘a’.

With empty lists, we just return the empty lists:

filter p [ ] = [ ]

With non-empty lists:

p (x xs) = if p x then x ( filter p x s)

else filter p xs

onlyCaps = filter isCap

noCap = filter (\x -> not (isCapx))

The \ is an anonymous function break.

* + - 1. What if you want to apply a function al all list elements?

[1, 2, 3] +1 to each element 🡪 [ 2, 3, 4]

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

This is more general to allow them to be different!

Define map by recursion:

map f [ ] = [ ]

map f (x xs) = (f x ) : (map f xs)

map (1n -> n +1) [ 1, 2, 3]

= (1n -> n +1) 1) (map (1n -> n+1) [2, 3])

= 2 (map (1n -> n+1) [2, 3] )

= 2 (3 (map (1n -> n+1) [3]))

= 2 (2 (4 (map (1n -> n+1) [ ] )))

= 2 (3 (4 I ) ) ) = [2, 3, 4]

* + - 1. Summing a list? Or some other aggregation? Map won’t do this so you need something more general.
      2. Folding lets you perform an aggregation operation.
         1. Two different kinds of fold: foldr, foldl
         2. Foldr (right):

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

foldr f v [ ] = v

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

f xo (f x1 (f x2 v) )

x0 : (x1: (x2 : [ ] ))

* + - 1. Suppose we wanted to sum all elements of the list:

sum [Int] -> Int

sum xs : foldr (+) 0 xs

Example: sum (2, 4, 6) = foldr (+) 0 [ 2, 4, 6]

= (+) 2 (foldr (+) 0 [4, 6] )

= 2 + (foldr (+) 0 [4, 6] )

= 2 + (4 + (foldr (+) 0 [6] ))

= 2 + (4 + (6+ (foldr (+) 0 [4, 6] ))

= 2 + (4 + (6 + 0) ) = 12

##end notes##