COURSE NAME: CS558

DATE: 8/23/18

1. Lecture
   1. Professor started late.
   2. Last time, we reviewed the syllabus and started discussing Haskell.
      1. Professor will load his handwritten notes from lecture onto Learn.
      2. Last time we covered: Int Bool Float Double
         1. Defining functions is crucial in HP.

square : : Int -> Int

square ~ = ~ \* ~

* + - 1. We can use these functions to define further functions.
         1. Last time we looked at the fourthpower function.
    1. Recursion is crucial in HP. We looked at that as a factorial last time.

fact Integer -> Integer

* + - 1. We would usually define this by iteration in C/Java. But here we do it by recursion.
      2. This is a recursive function because it is defined in terms of itself.
      3. The other way to define this: use if/and/else

fact n = if n : : 0 then ! else n & fact ( n-1)

* + - 1. This depends on personal taste and style.
         1. Professor wrote additional examples.
    1. If you want to create more than 1, use a list. Lists are key building blocks in HP.
       1. You can explicitly spell the contents:

[1, 2, 3, 4]

* + - 1. Built up internally using Empty List

Empty list [ ]

* + - 1. Then add the element by cons :

So this list is actually 1: (2 : (3: (4: [ ] ) ) )

* + - 1. This is the type for lists that contain integers.
      2. You can also nest them!
    1. Using length:

length: : [a] -> Int

length [ ] = 0

* + - 1. The head of the list is the beginning. It has type Int. The tail is the remainder and it has type [Int]
      2. Professor provided an example for evaluating Length of [1,2].
      3. The [a] in the example above is a type variable.
         1. When you have more than 1 type variables, you get a polymorphic type.
         2. This means you can use the length function for any time like this (a list of something that you don’t know right now).
         3. So you can use length for many purposes!
    1. Strings
       1. The string type is a synonym for a list of characters.

“abc” = = [‘a’, ‘b’, ‘c’,]

* + 1. Head element of lists

head : : [a] -> a

Empty and non empty types:

head [ ] +

head [x: xs) = x

* + - 1. The first is an empty list. So add error:

head [ ] = error “empty list”

* + - 1. You can also write as:

(head [ ] = error “empty list”)

* + - * 1. You might get an error because it doesn’t cover all cases.
    1. Tail of list:

tail : : [a] -> [a]

tail [ ] = error “empty”

tail (x:xs) = xs

* + 1. What if you want to get the first “n” elements of a list?

take 3 (1: (2: (3: (4: [ ] ) ) ) ) = [1, 2, 3]

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

* + - 1. How many things do you have left to take? When you hit zero, you want to stop. Or you may run out of things before. So there are three cases:

take 0 xs = [ ]

take n [ ] : [ ]

take n (x, xs) = x : take (n – 1) xs

Professor finished what would happen. Please see his notes for completed code.

* + 1. What if we wanted to sum the list of integers?

sum : [Int} -> Int

sum [ ] = 0

sum (x, xs) = x + sum xs

* + - 1. This version is too restricted:

[Int] -> Int

* + - 1. This version is too permissive:

Sum [a ] -> a

* + - 1. Use this:

sum : : (Num a) => [a] ->a

* + - * 1. What we are saying 🡪 The sum has this type for a such that …
    1. A few of the type classes that you would see: Eq (= = ), Ord ( less than, greater than), ones = 1: ones
    2. Suppose we have a function that is a list of value f(x) for x from 1 to 10. How to write?

[f x | x <- [1, 10] ]

* + - 1. What about multiple indices:

[g x y | x <- [1, 10], y < - [1, ,, 100]

* + 1. Getting a specific element from a list:

(!!) … [a] -> Int -> a

[ ] !! n = error “empty”

(x, xs) !! 0 = x

(x, xs) !! = xs !! (n-1)

##end notes##