Week 4

Ch 5: Higher-Order Functions
Ch 6: Modules

University of the Fraser Valley

Dr. Russell Campbell

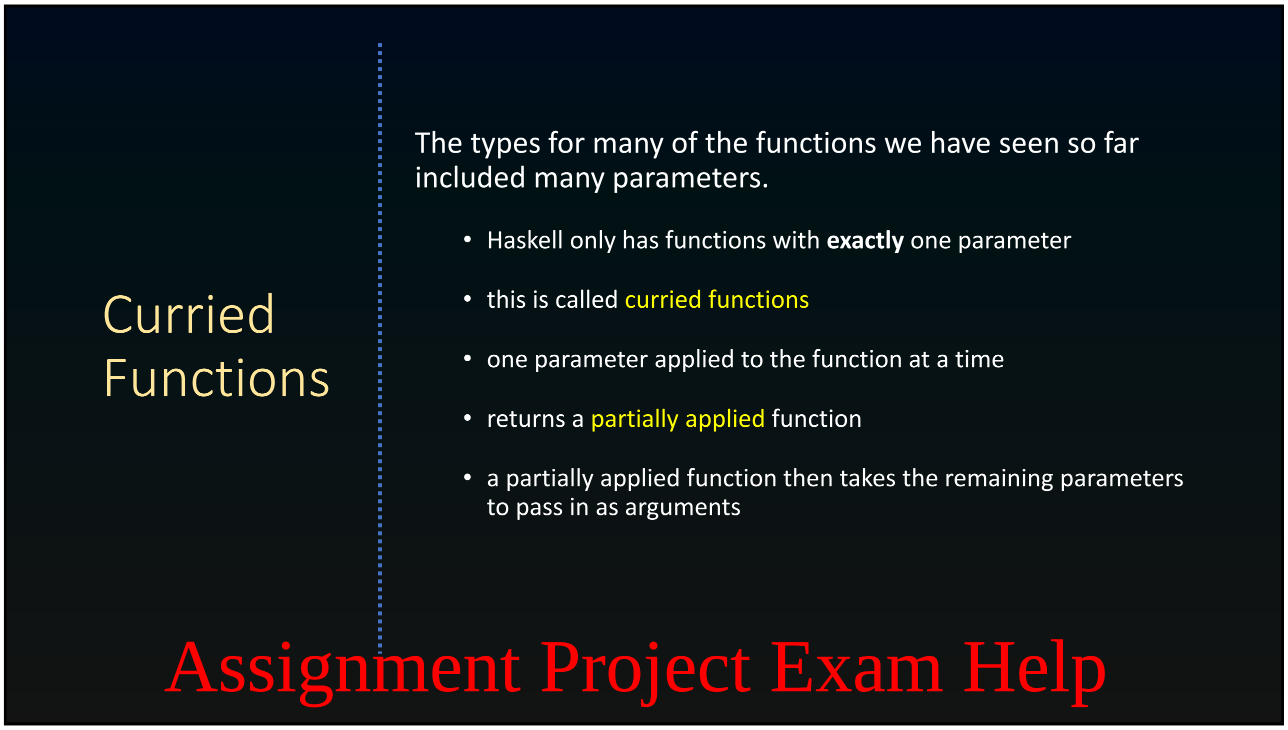
Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

1

Overview

2

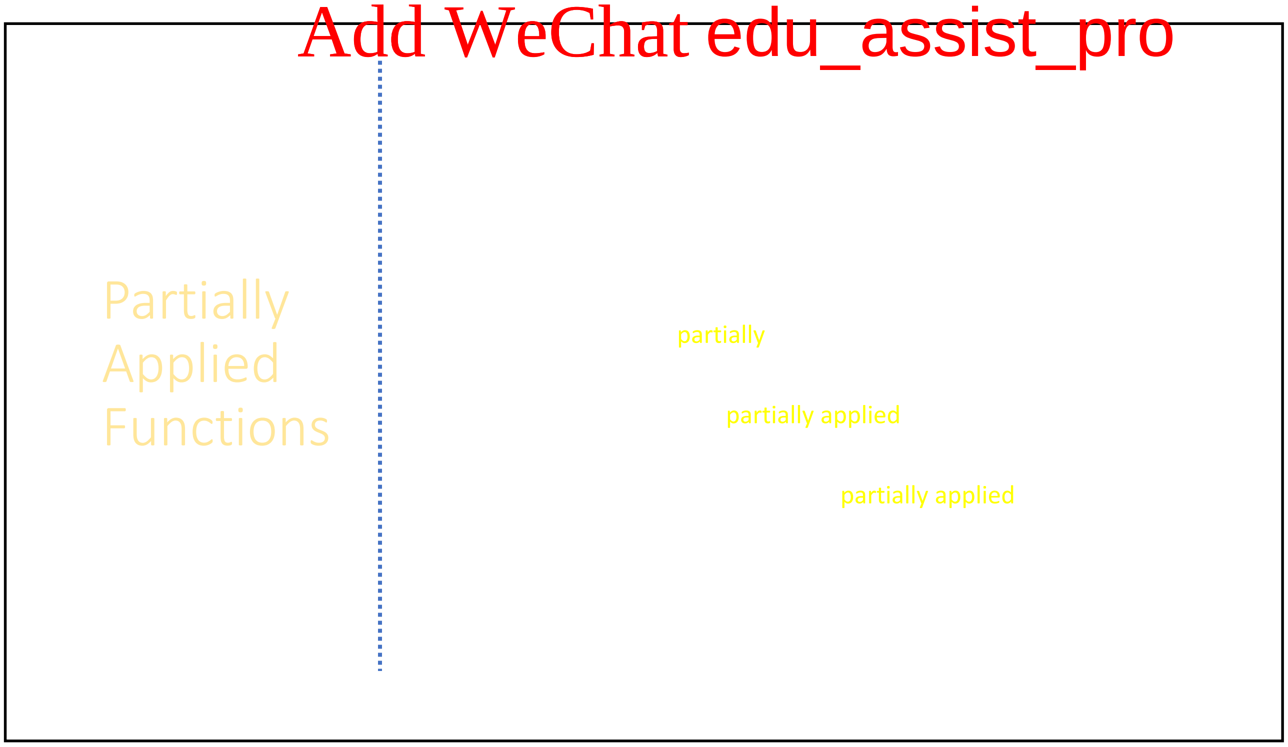


Curried Functions

The types for many of the functions we have seen so far included many parameters.

- Haskell only has functions with **exactly** one parameter
- this is called curried functions
- one parameter applied to the function at a time
- returns a partially applied function
- a partially applied function then takes the remaining parameters to pass in as arguments

3



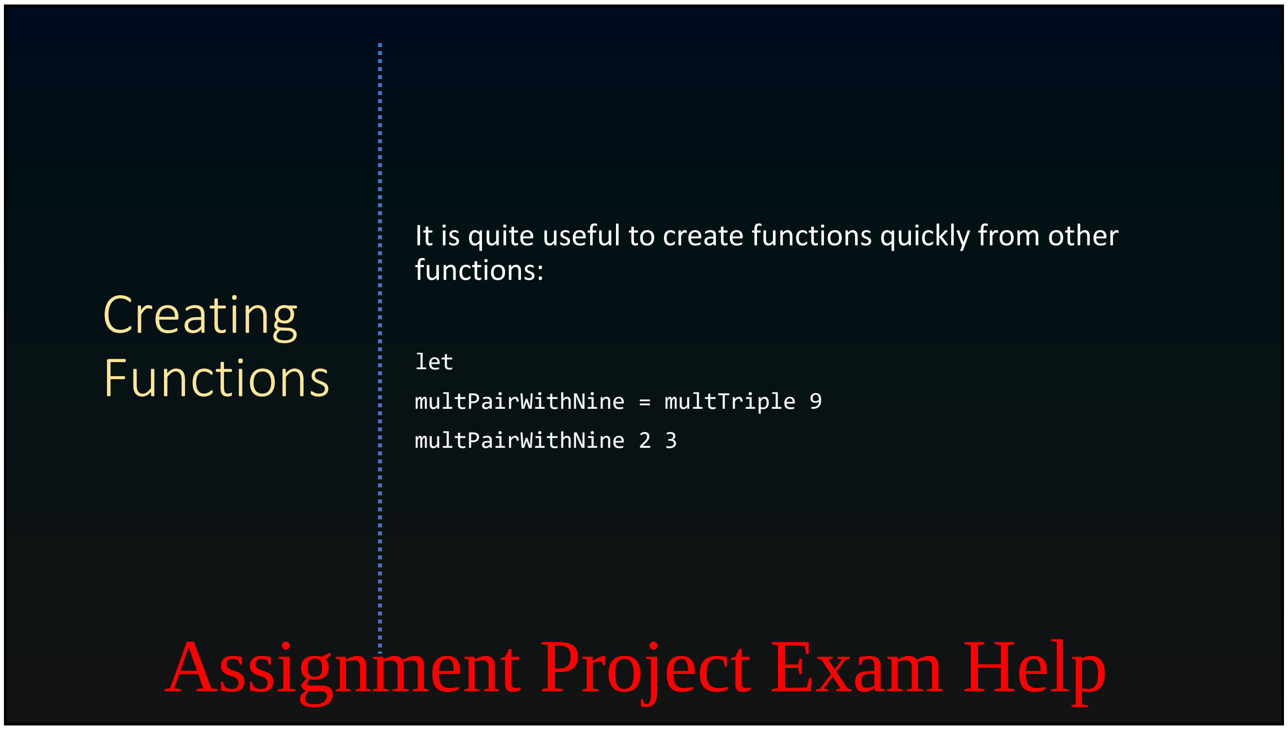
Partially Applied Functions

partially

partially applied

partially applied

4

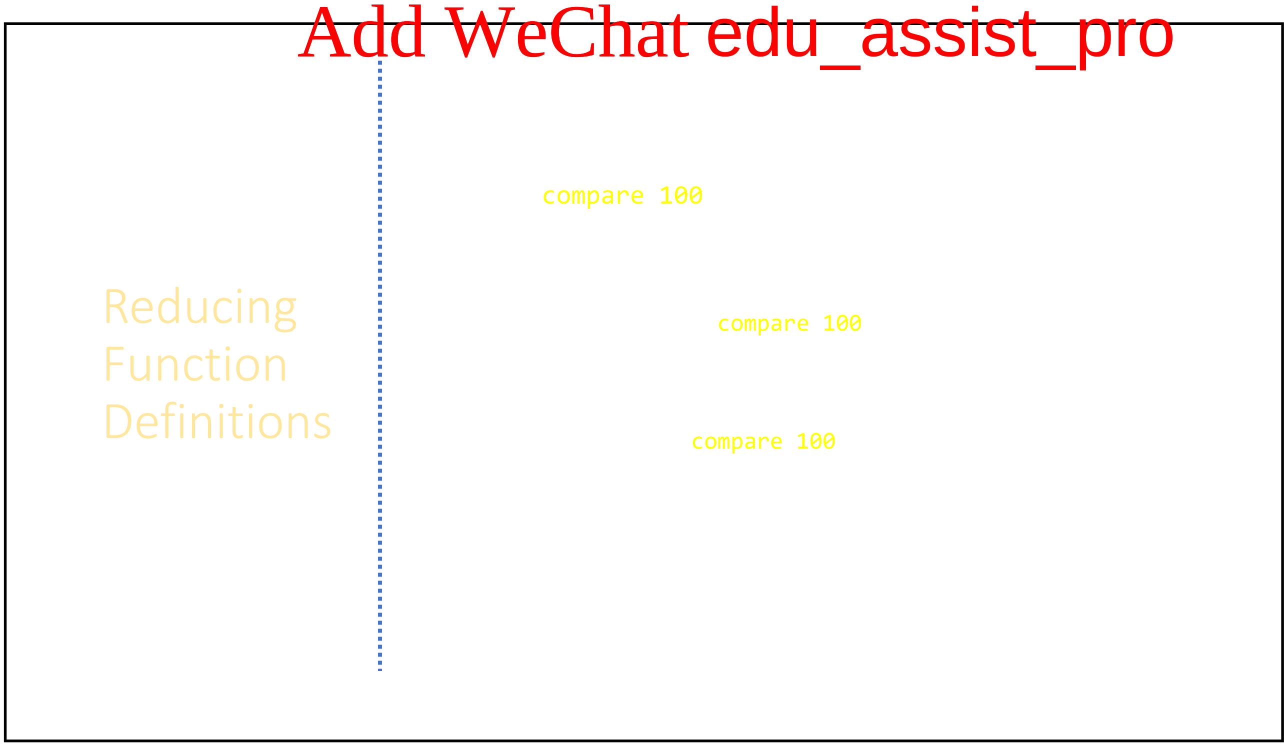


Creating Functions

It is quite useful to create functions quickly from other functions:

```
let
multPairWithNine = multTriple 9
multPairWithNine 2 3
```

5



Reducing Function Definitions

```
compare 100
```

```
compare 100
```

```
compare 100
```

6

Parentheses Around Functions

Sections are functions surrounded by parentheses:

```
(/10) 20
```

```
isUpper :: Char -> Bool
isUpper = (`elem` ['A'..'Z'])
```

The minus sign has multiple uses, where `(-1)` means a negative number, not partially applied subtraction:

- partially applied subtraction function is `(−)`
- OR e.g.: `(subtract 1)`
  "subtract 1 from the next parameter"

7

Cannot Print Functions

```
No instance for (Show (Int -> Int)) arising from a use of `print’
(maybe you haven't applied a function to enough arguments?)
In a stmt of an interactive GHCi command: print it
```

8

Functions as Parameters

Haskell can define functions that take other functions as parameters.

```
let
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

Try the examples of using the `applyTwice` function:

```
applyTwice (+3) 10
applyTwice (++ " HAHA") "HEY"
applyTwice ("HAHA " ++) "HEY"
applyTwice (multTriple 2 2) 9
applyTwice (3:) [1]
```

9

zipWith'

```
f x y f x y
```

10

Flipping Parameters

Consider the two ways to implement the standard library `flip` function:

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x

flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f x y = f y x
```

Note that because functions are curried, the second set of parentheses in the type description is not needed.

11

zipWith and flip'

flip'

flip'

12

Processing Lists

Most of the advantages of functional programming use operations on lists:

```
map' :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs
```

```
map' (replicate 3) [3..6]
```

```
map' (map' (^2)) [[1,2],[3,4,5,6],[7,8]]
```

```
map' fst [(1,2),(3,4),(5,6),(7,8),(9,10)]
```

13

map

map

map

map

14

Filtering Lists

A predicate is a function:

- that inputs something
- and results in `True` or `False`

The `filter` function applies a predicate function

- to the elements of a list
- and creates a new list with only those elements that return `True` by the predicate.

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' p (x:xs)
 | p x    = x : filter' p xs
 | True   = filter' p xs
```

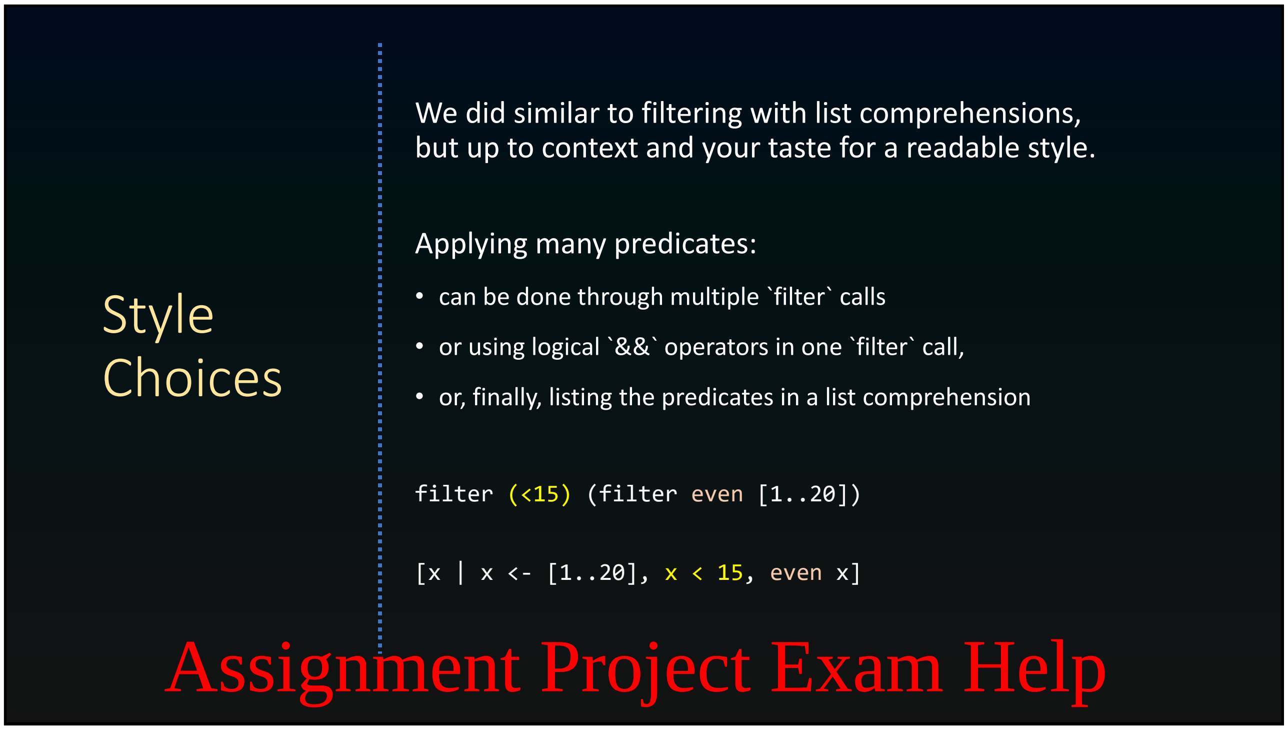
15

filter Examples

filter

filter

16



Style Choices

We did similar to filtering with list comprehensions, but up to context and your taste for a readable style.

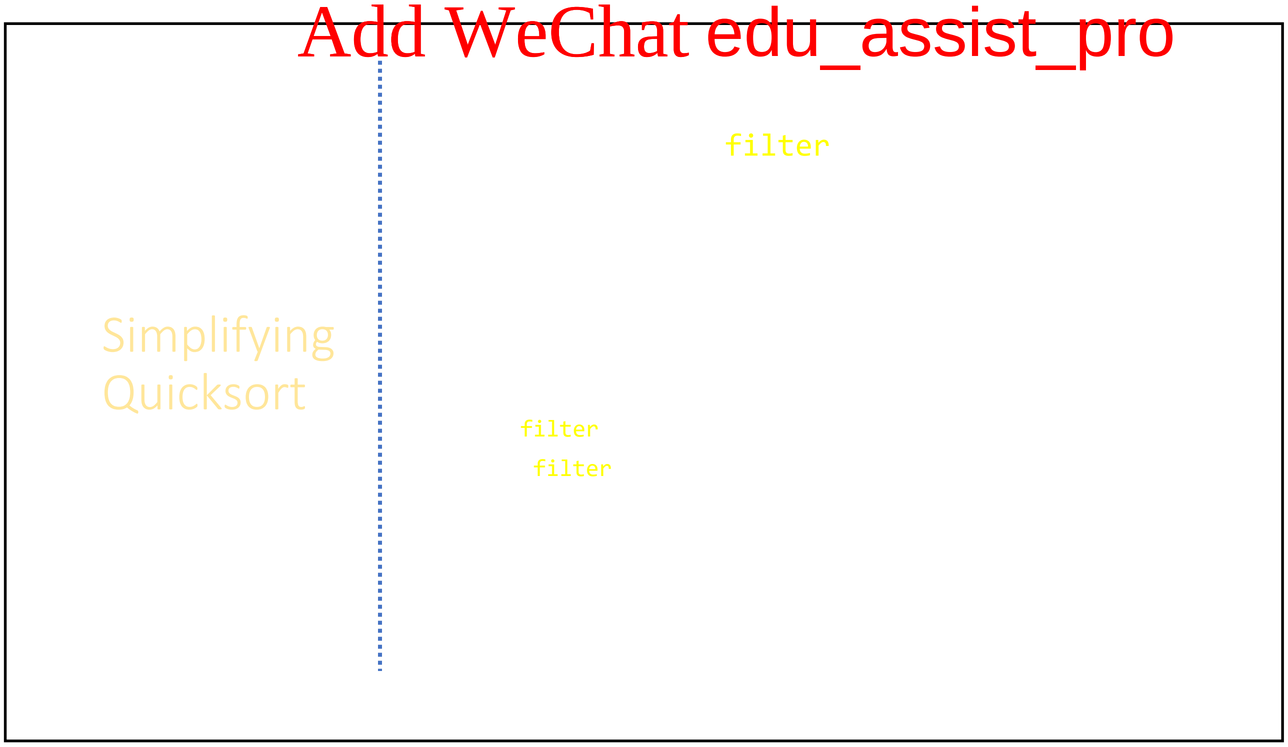
Applying many predicates:

- can be done through multiple `filter` calls
- or using logical `&&` operators in one `filter` call,
- or, finally, listing the predicates in a list comprehension

```
filter (<15) (filter even [1..20])

[x | x <- [1..20], x < 15, even x]
```

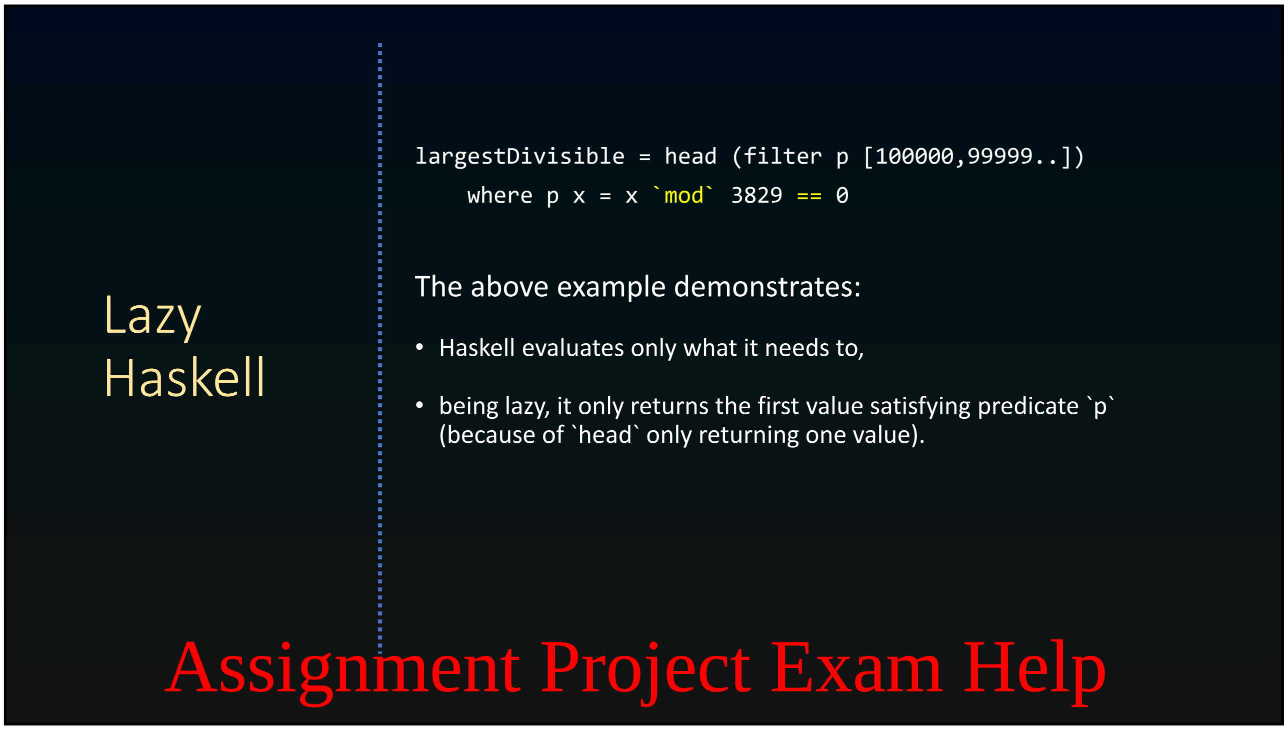
17



Simplifying Quicksort

```
filter
filter
 filter
```

18



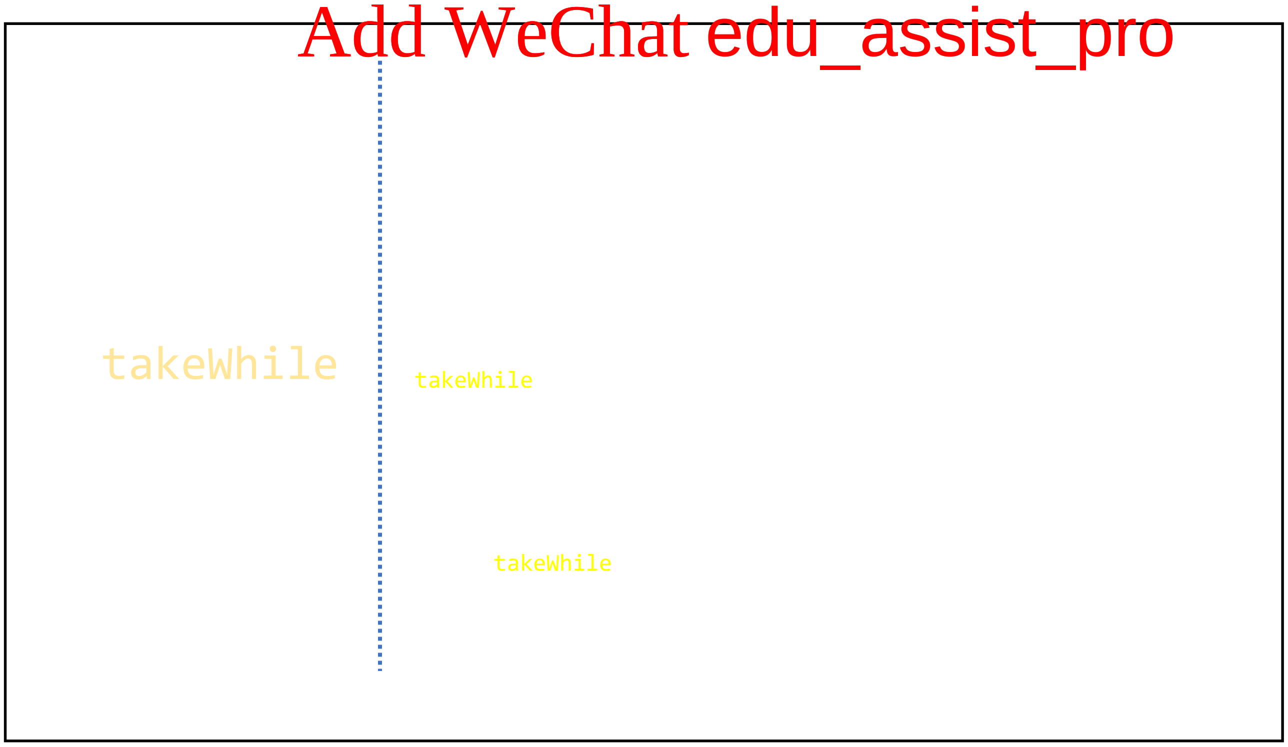
Lazy Haskell

```haskell
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

The above example demonstrates:

- Haskell evaluates only what it needs to,
- being lazy, it only returns the first value satisfying predicate `p` (because of `head` only returning one value).

19



takeWhile

takeWhile

takeWhile

20

Alternative to Nesting

We can rearrange how function composition is written using the concept of piping:

```
let a `pipe` b = flip ($) a b

:{
    (map (^2)) [1..]
    `pipe`
    (filter (odd))
    `pipe`
    (takeWhile (<10000))
    `pipe`
    (sum)
:}
```

21

But Wait, There's More…

`odd (m^2)`

22

Collatz Chains

- begin with any natural number (1 or larger) for a_0
- if a_{n-1} is 1, stop, otherwise:

$$\begin{cases} \frac{1}{2}a_{n-1}, & a_{n-1} \text{ even} \\ 3a_n + 1, & a_{n-1} \text{ odd} \end{cases}$$

- repeat with the resulting number

Does the sequence that forms a chain always end in the number 1?

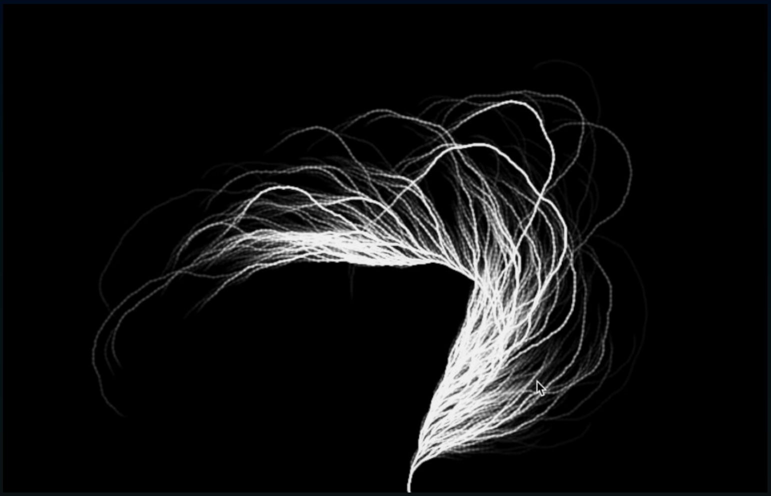
- this is an open problem no one has solved yet
- the largest value known to stop at 1 is 2^100000 - 1 (as of 2018)
    - https://ieeexplore.ieee.org/document/8560077

23

Collatz Chain

24

Collatz Conjecture



- copyright: (YouTube) Coding Train
- Coding in the Cabana 2: Collatz Conjecture
- https://youtu.be/EYLWxwo1Ed8?t=1263

25

How Many Long Chains?

26

Making More Functions

Partially applied functions can be used to create functions that take multiple parameters.

Combine this with Haskell's laziness:

```
listOfFuns = map (*) [0..]
(listOfFuns !! 4) 5
```

- the first line returns a function for each element
- the last line pulls element at index `4` to apply its function `(4*)` to the value `5`

27

Lambda Functions

lambda

```
(filter (\xs -> length xs > 15)
```

28

Function Styles

Use partial application to avoid using lambdas when it is not even necessary, as seen in equivalent examples:

```
map (\x -> x + 3) [1, 2, 3]
map (+3) [1, 2, 3]
```

A few more involved examples with two parameters:

```
zipWith (\a b -> a + b) [6,5,4,3,2,1] [1,2,3,4,5,6]
```

Another example with pattern matching:

```
map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
```

29

Cases, and Runtime Error

`case`

`case` `of`

`(1, 2)`

30

Emphasized Currying

This example overemphasizes currying by way of unnecessary lambda functions:

```
addThree' :: Int -> Int -> Int -> Int
addThree' x y z = x + y + z

addThree' :: Int -> Int -> Int -> Int
addThree' = \x -> \y -> \z -> x + y + z
```

Note that lambdas written without parentheses assume everything to the right of `\` belongs to the lambda.

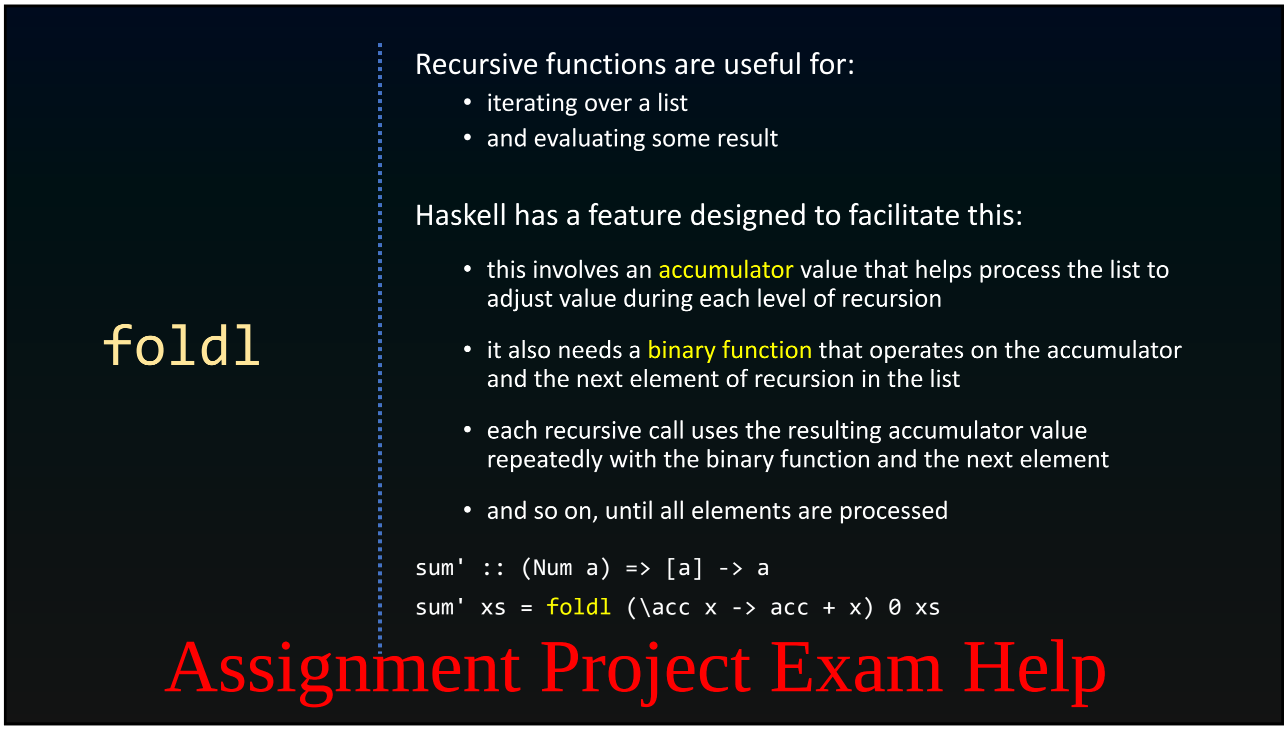
31

`flip` Implementation

`flip`

`flip`

32



foldl

Recursive functions are useful for:

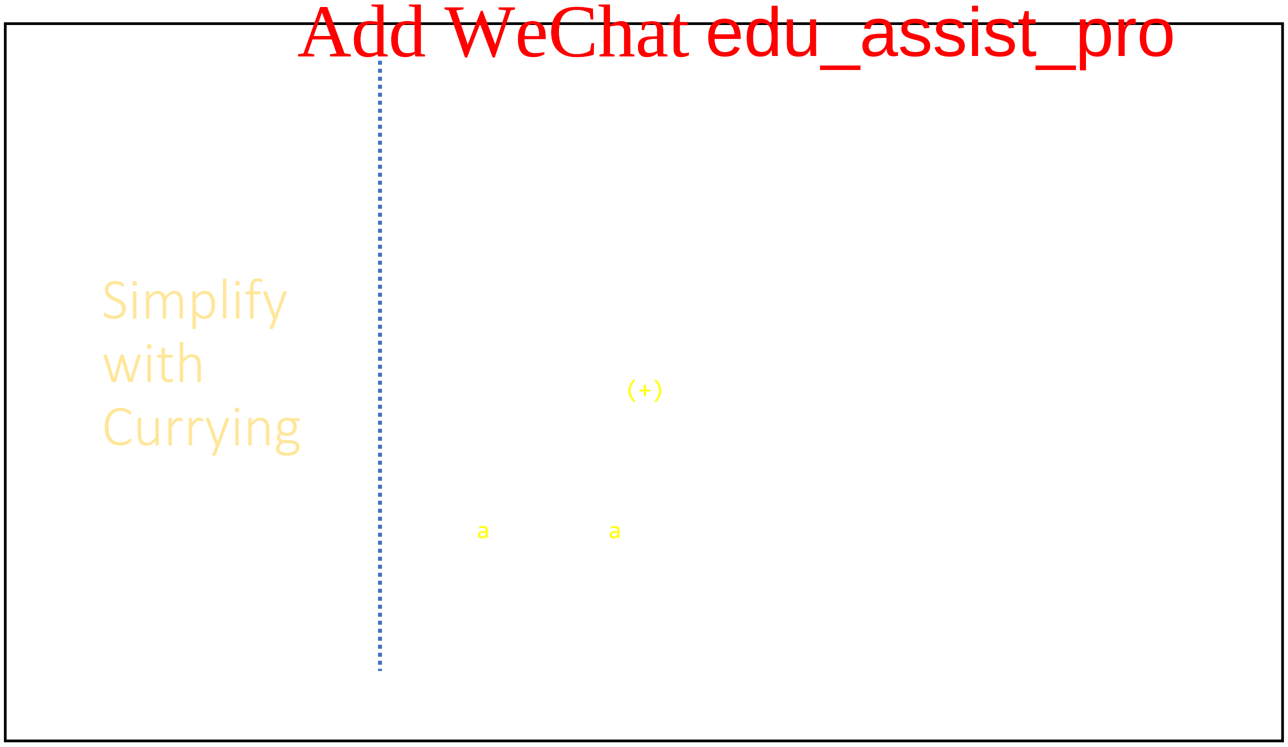
- iterating over a list
- and evaluating some result

Haskell has a feature designed to facilitate this:

- this involves an accumulator value that helps process the list to adjust value during each level of recursion
- it also needs a binary function that operates on the accumulator and the next element of recursion in the list
- each recursive call uses the resulting accumulator value repeatedly with the binary function and the next element
- and so on, until all elements are processed

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

33

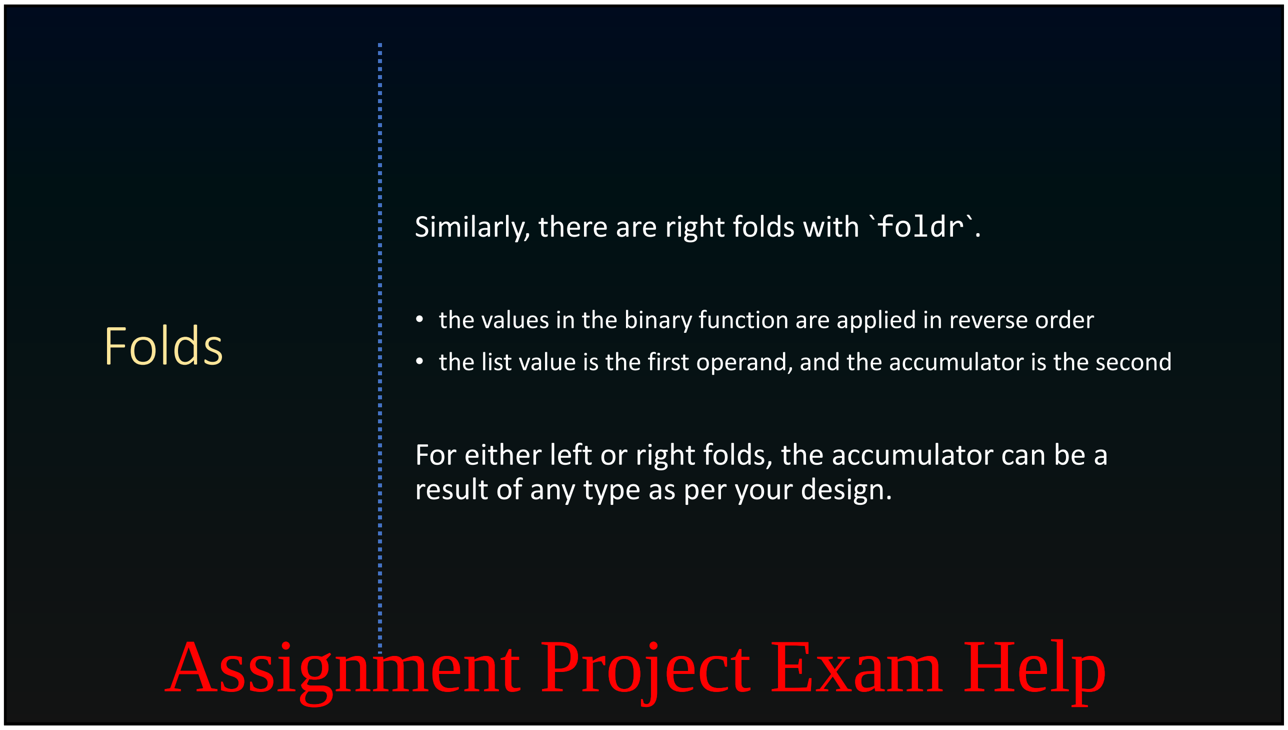


Simplify with Currying

(+)

a a

34



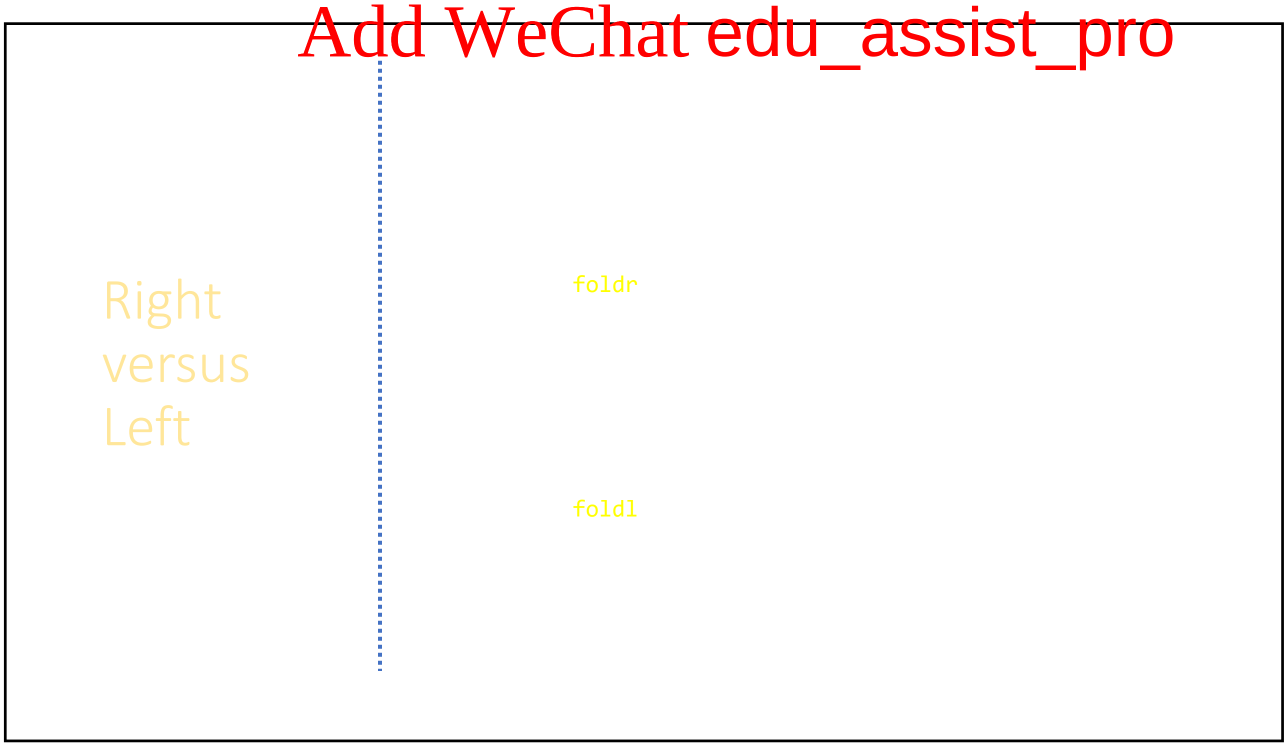
Folds

Similarly, there are right folds with `` `foldr` ``.

- the values in the binary function are applied in reverse order
- the list value is the first operand, and the accumulator is the second

For either left or right folds, the accumulator can be a result of any type as per your design.

35



Right versus Left

`foldr`

`foldl`

36

`++` versus `:`

The `:` operation is more efficient than `++` operation, so we mostly use folding on the right for processing lists.

Right folds work on infinite lists, whereas left folds do not.

37

elem

38

First Element Starts Accumulation

`foldl1` and `foldr1` functions assume the accumulator is the value of the first item in the list that they process.

Another implementation of `max`:

```
max' :: (Ord a) => [a] -> a
max' = foldl1 max
```

- partial application to help create functions
- the difficulty is in knowing that `foldl1` takes two parameters
- the second parameter is a list

39

Empty List or Not

40

— Break —

41

```
reverse'
```
and
```
product'
```

42

filter

Another implementation of `` `filter` ``:

```
filterr :: (a -> Bool) -> [a] -> [a]
filterr p = foldr (\x acc -> if p x then x : acc else acc) []
```

Remember, the accumulator parameter is always ordered on the side you are folding.

43

last

44

Right Fold Nesting Operations

Suppose we want to apply a right fold with binary function `f` on the list `[3,4,5,6]`.

- this can be seen as the expression

`f 3 (f 4 (f 5 (f 6 acc)))`

- the value `acc` is the starting accumulator value

- if `f` is replaced with `+` and `acc` starts with `0`,
- then the expression would be

`3 + (4 + (5 + (6 + 0)))`.

45

Left Fold Nesting Operations

46

Infinite? Use `foldr`

The `and'` function will combine Boolean elements of a list together with the `&&` operator.

```
and' :: [Bool] -> Bool
and' = foldr (&&) True
```

Take special care to use the `foldr` function, and not the `foldl` function, since the input could be an infinite list.

47

Short Circuiting

short circuits evaluation

48

scan

`scanl`/`scanr` functions are similar to `foldl`/`foldr`

- they return a list with all of the intermediate accumulator values from processing the input list
- of course, there are also the `scanl1` and `scanr1` functions

49

scan

50

Example with Piping

```
sqrtSums :: Int
sqrtSums = length (takeWhile (<1000)
    (scanl1 (+) (map sqrt [1..])))
```

Equivalently:

```
sqrtSums :: Int
sqrtSums =
 (map sqrt) [1..]
 `pipe` (scanl1 (+))
 `pipe` (takeWhile (<1000))
 `pipe` (length)
```

```
sum (map sqrt [1..130])
sum (map sqrt [1..131])
```

51

Using $ to Apply Functions

```
) -> a
```

52

Replace $ for Parentheses

Observe how $ can clean up nesting a bit:

```
sum (filter (> 10) (map (*2) [2..10]))

sum $ filter (> 10) (map (*2) [2..10])

sum $ filter (> 10) $ map (*2) [2..10]
```

53

Alternative Syntax

54

Apply Functions with $ as a Parameter

Another important use of $ is to tell Haskell to immediately apply some function:

```
map ($ 3) [(4+), (10*), (^2), sqrt]
```

Note that the `($ 3)` function takes some other function as input and applies that function to `3`.

55

Function Composition

56

Right Associative

- function composition is right-associative
- this is similar to our right folds, with

```
f (g (h x))
```

equivalently

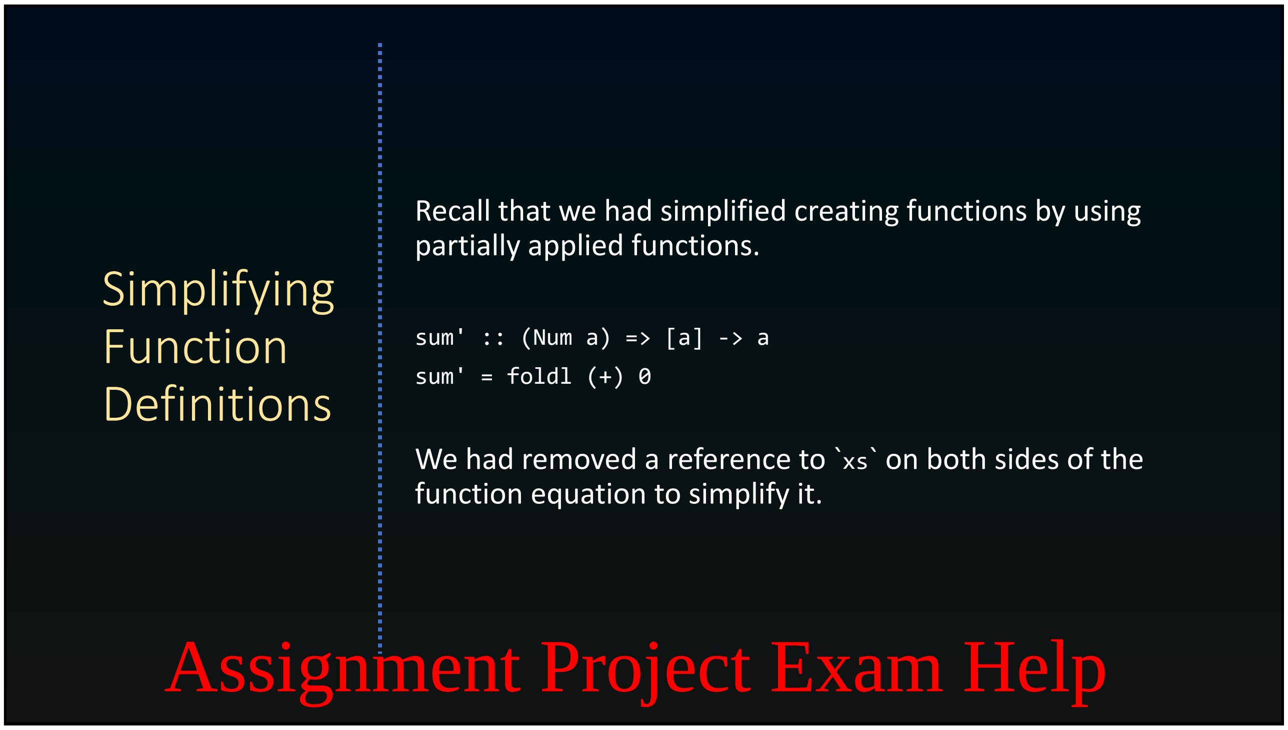
```
(f . g . h) x
```

```
map (negate . sum . tail) [[1..5],[3..6],[1..7]]
```

57

Currying and Multiline: . and $

58



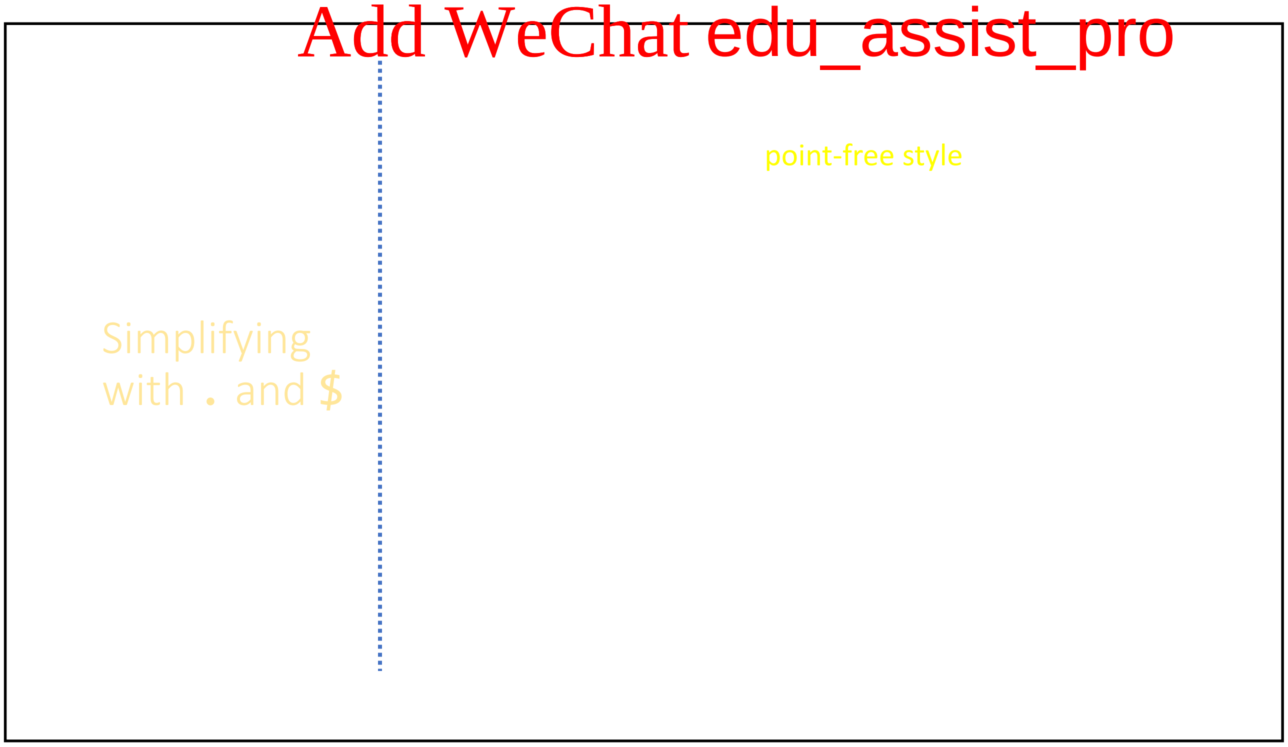
Simplifying Function Definitions

Recall that we had simplified creating functions by using partially applied functions.

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

We had removed a reference to `xs` on both sides of the function equation to simplify it.

59



Simplifying with . and $

point-free style

60

Point-free Form

Now we rewrite `sqrtSum`

- sum of square roots for the first `n` integers reaches a threshold of `1000` in point-free form:

```
sqrtSum :: Integer
sqrtSum = length . takeWhile (<1000) . scanl1 (+) $ map
(sqrt) [1..]
```

Alexis King demonstrates extreme consideration of reduction optimizations in realistic code:

- https://www.youtube.com/watch?v=yRVjR9XcuPU
- current research!

61

— Break —

62

— Chapter 6 —

63

Modules

exported

reuse

64

import

The Haskell standard library is separated by modules, e.g.:

- managing lists
- concurrent programming
- complex numbers
- and more…

The type classes, types, and functions we have used are part of the default imported Prelude module.

To import modules:

```
import Data.List
```

65

Data.List Module Functions

nub

66

Import Variations

To import in an interactive ghci session:

```
:m + Data.List
```

We can specify which functions we want to use:

```
import Data.List (nub, sort)
```

Or those we do not want, to avoid naming conflicts:

```
import Data.List hiding (nub)
```

67

Naming Conflicts

```
qualified
```

68

Function Composition vs Module Accessor

The previous examples show how to avoid

- the `Data.Map.filter` and `Data.Map.null` functions
- do not conflict with the default `Prelude.filter` nor the `Prelude.null` functions.

Note that the dot operator is used here to access the function from the module and is not function composition:

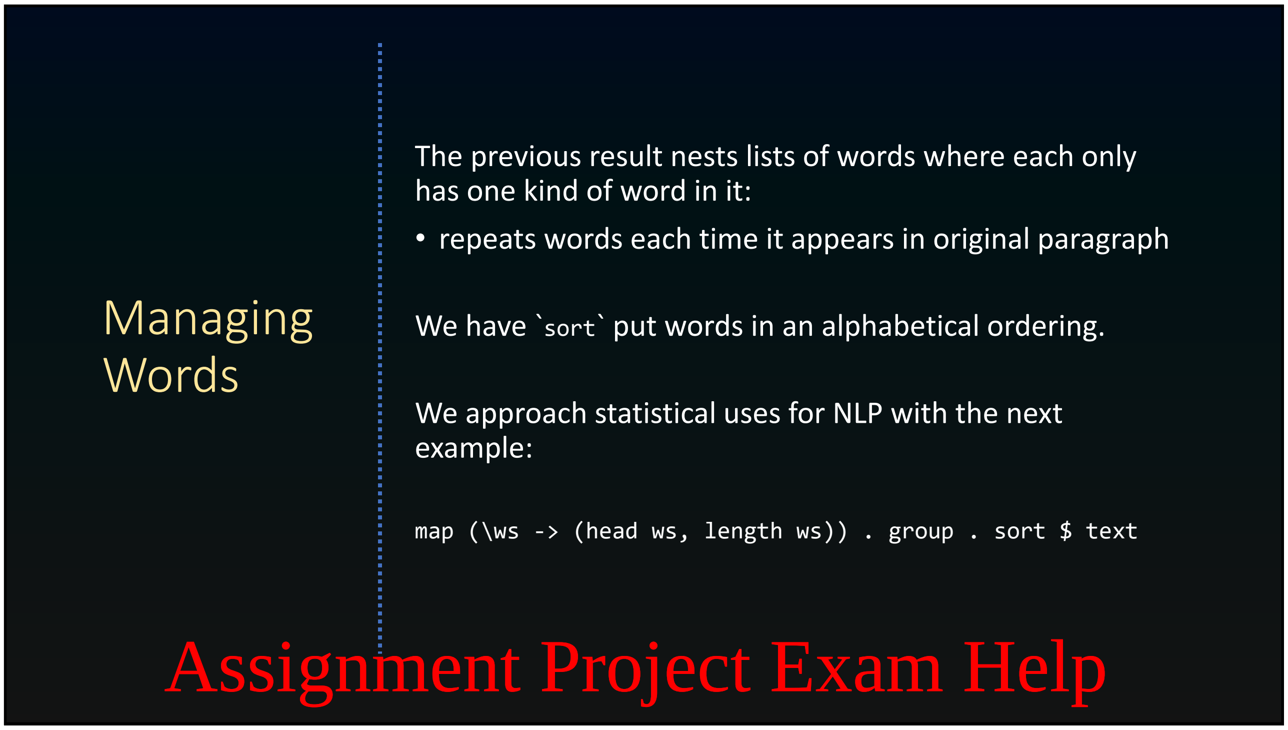
- ***make sure*** to not use any whitespace before nor after the dot
- use a space before and after a function composition dot ` . `

69

words

Working with Text

70



Managing Words

The previous result nests lists of words where each only has one kind of word in it:

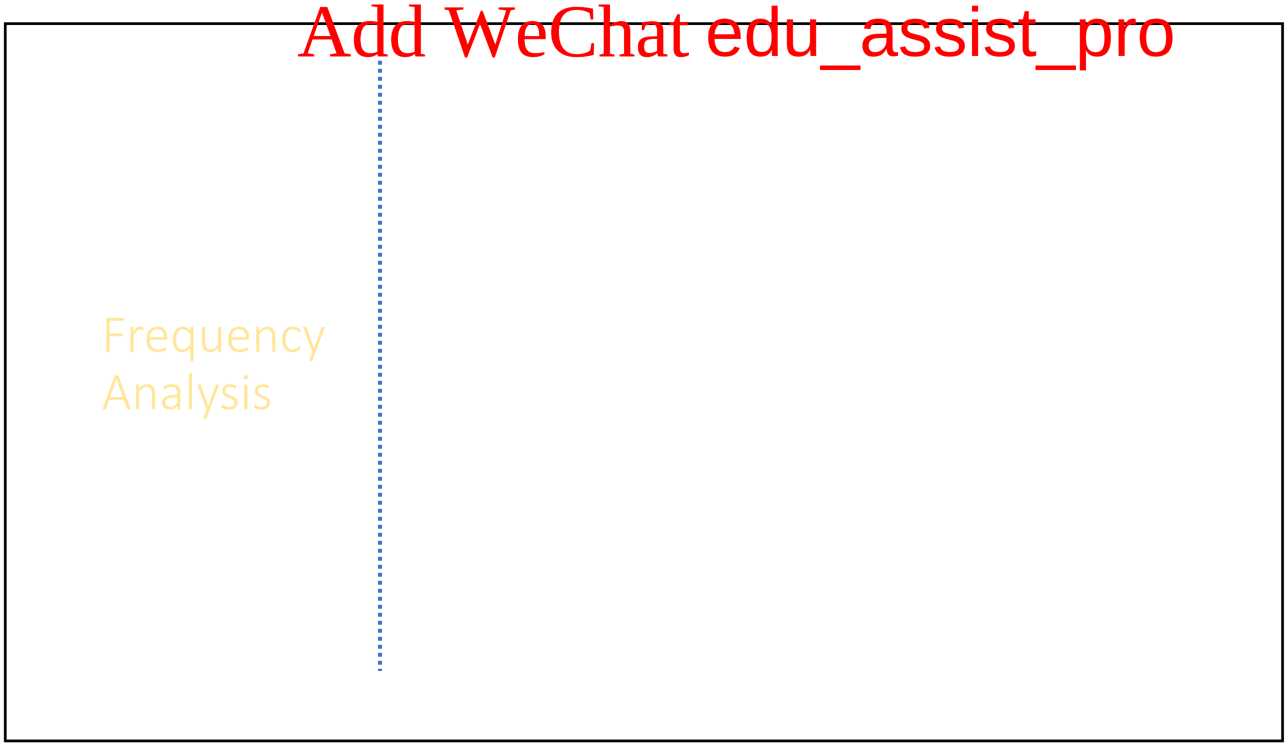
- repeats words each time it appears in original paragraph

We have `` `sort` `` put words in an alphabetical ordering.

We approach statistical uses for NLP with the next example:

```
map (\ws -> (head ws, length ws)) . group . sort $ text
```

71



Frequency Analysis

72

Try the following statements:

```
tails "party"
"ha" `isPrefixOf` "hawaii"
any (> 4) [1,2,3]
any (> 4) [1,2,3,4,5]
```

- `tails` folds a list of accumulated tail elements
- `isPrefixOf` tests for prefix of first argument at start of the second argument
- `any` will return `True` if at least one element of an input list satisfies the predicate function passed in

73

74

The `Data.Char` module with the `ord` and `chr` functions converts back and forth between numbers/letters:

```
ord `a`
chr 97
```

A short function to do the Caesar Cipher for us:

```
import Data.Char

let
caesar :: Int -> String -> String
caesar offset msg =
    map (\c -> chr $ (ord c + offset) `mod` 26)) msg
```

75

76

Careful with Left Folds

Possibility of stack overflows with using `foldl`:

```
foldl (+) 0 (replicate 100000000 1)
```

Order of operations plays out something like the following:

```
foldl (+) 0 [1,2,3] =
foldl (+) (0 + 1) [2,3] =
foldl (+) ((0 + 1) + 2) [3] =
foldl (+) (((0 + 1) + 2) + 3) [] =
((0 + 1) + 2) + 3 =
(1 + 2) + 3 =
3 + 3 =
6
```

77

Avoiding Deferral

78

No Stack Overflow

Then it would be safe to try:

```
foldl' (+) 0 (replicate 100000000 1)
```

There are similar versions for the other fold functions.

79

find

Maybe a

Maybe a

Nothing

Just a

a

find

80

Data.Char

We will use `find` to help us sum the digits of numbers.

- consider finding the smallest integer with digits that add up to `40`
- we also use `digitToInt` function from the `Data.Char` module:

```
digitToInt '2'
```

Convert hexadecimal digits to decimal:

```
import Data.Char
import Data.List

digitSum :: Int -> Int
digitSum = sum . map digitToInt . show
```

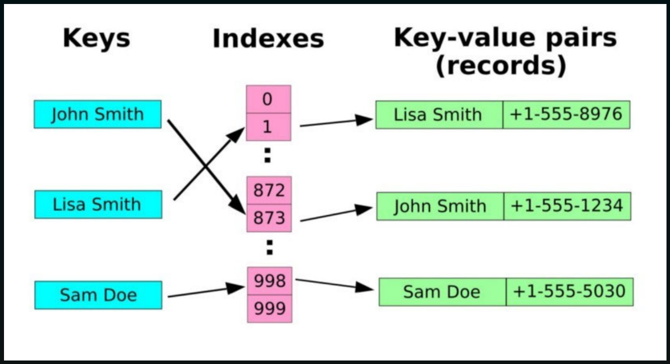
81

Get What We Want Quickly

82

Map Data Structure

Data structure for a map is similar in Haskell to what they call an association list. (we are not implementing with hash table)



source: vinodronold (Medium, 2020)

https://medium.com/@vinodronold/hash-tables-implementation-in-peoplecode-231bca18442d

83

Association List:

Word Keys and Phone Number Data

84

Get Data from Our Association List

Function to extract one of the values matching input key:

```
:m Data.Maybe Data.List

let
findKey :: (Eq k) => k -> [(k, v)] -> v
findKey key xs =
    snd (
        fromJust $ find (\p -> (fst p) == key) xs
    )
```

85

Alternative to Get Data

86

Maybe

We use `Maybe` to deal with the case when there is no matching key (...but not quite like exceptions).

```
findKey :: (Eq k) => k -> [(k, v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs)
 | key == k  = Just v
 | True      = findKey key xs

findKey :: (Eq k) => k -> [(k, v)] -> Maybe v
findKey key xs = foldr (\(k, v) acc ->
        if k == key
            then Just v
            else acc
    ) Nothing xs
```

87

Context of List vs Elements

88

Data.Map

The `Data.Map` module has association lists:

- are efficient with many functions for managing them.
- but, there are many naming clashes with `Prelude` and `Data.List`
- so import with qualified:

```
import qualified Data.Map as Map
```

The `fromList` function turns an association list into a Map:

```
Map.fromList [(3,"shoes"),(4,"trees"),(9,"bees")]
Map.fromList [("MS",1),("MS",2),("MS",3)]
```

See how extra duplicate-key pairs are discarded.

- the result is displayed as a list, but with prefix `fromList`

89

Maps vs Association Lists

90

Example Map

Setup the phone book we created earlier as a map:

```
import qualified Data.Map as Map

let
phoneBook :: Map.Map String String
phoneBook = Map.fromList
 [("betty", "555-2938")
 ,("bonnie", "452-2928")
 ,("patsy", "493-2928")
 ,("lucille", "205-2928")
 ,("wendy", "939-8282")
 ,("penny", "853-2492")
 ]
```

91

Create New Maps

92

Map.size

A basic function to get the number of pairs in a map:

- `` `:t Map.size` `` has type `` `Map.size :: Map.Map k a -> Int` ``

Go ahead and give the function a try.

93

Convert Map Types

digitToInt Data.Char

94

Organizing Data

There could be duplicate phone numbers in our collection for a person with the same name.

- instead, we can accumulate values that correspond to the same key
- we can accumulate the values in a way we specify as parameter of the `fromListWith` function (which we see soon)

95

Repeated Keys with More Data

96

Collect by Key

```
findKey "patsy" phoneBook
findKey "wendy" phoneBook
findKey "betty" phoneBook
```

We could arrange multiple phone numbers per name:

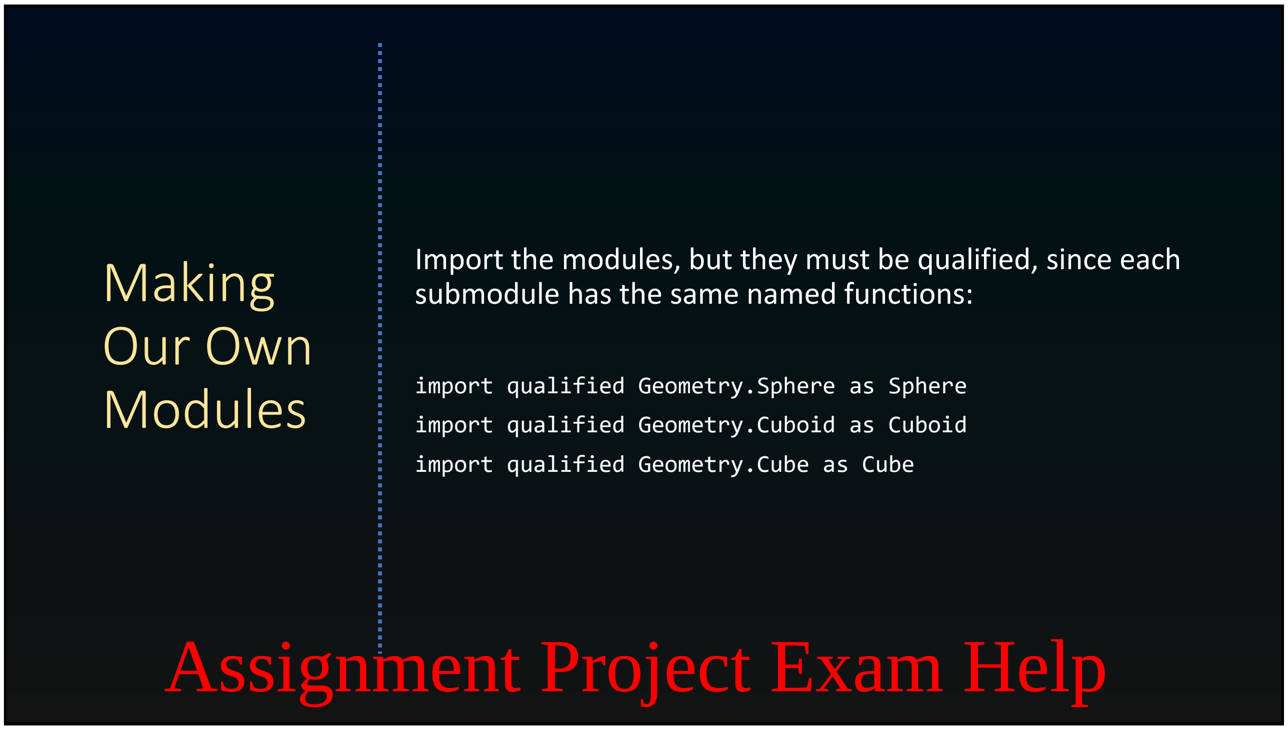
```
phoneBookToMap :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs =
    Map.fromListWith (++) $ map (\(k, v) -> (k, [v])) xs

Map.lookup "patsy" $ phoneBookToMap phoneBook
Map.lookup "wendy" $ phoneBookToMap phoneBook
Map.lookup "betty" $ phoneBookToMap phoneBook
```

97

Querying Maps

98

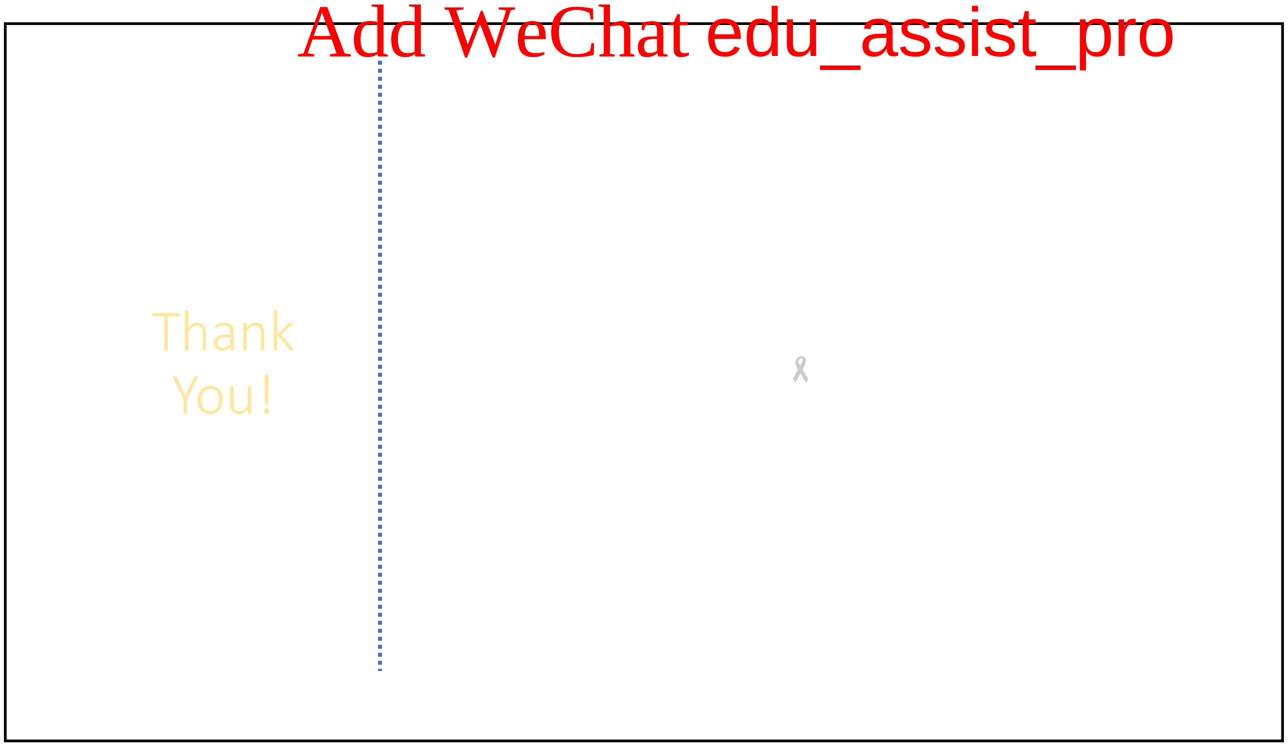


Making Our Own Modules

Import the modules, but they must be qualified, since each submodule has the same named functions:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

99



Thank You!

100