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

Assignment Project Exam Help

1

https://powcoder.com

Add WeChat powcoder

Overview

Chapter 5:

- function currying
- creating functions
- simplifying functions
- functions for processing with Lists
  - folds
  - filters
  - maps
- composition
- applying functions when passed as parameter

Chapter 6:

- modules
  - Data.List, Data.Char, Data.Maybe, Data.Map

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

Assignment Project Exam Help

3

https://powcoder.com

Add WeChat powcoder

Partially Applied Functions

```
multTriple :: Int -> Int -> Int
multTriple x y z = x*y*z
```

This function could be called with `multTriple 3 5 9`, but the expression `((multTriple 3) 5) 9` is equivalent.

- applies functions partially in order of nested parentheses, innermost first
- `multTriple 3` is a partially applied function with one constant `3` and takes two parameters
- also, `(multTriple 3) 5` is a partially applied function with constants `3` and `5` and takes one parameter
- note that the function type can be written equivalently as `multTriple :: Int -> (Int -> (Int -> Int))`

4

Creating Functions

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

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

Assignment Project Exam Help

5

https://powcoder.com

Add WeChat powcoder

Reducing Function Definitions

The following two function definitions are equivalent:

- because `compare 100` is a partially applied function itself

```
compareWithHundred :: Int -> Ordering
compareWithHundred x = compare 100 x
```

```
compareWithHundred :: Int -> Ordering
compareWithHundred = compare 100
```

Note that `compare` has type `(Ord a) => a -> (a -> Ordering)`

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"

Assignment Project Exam Help

7

https://powcoder.com

Add WeChat powcoder

Cannot Print Functions

Without `let`, if we just type a partially applied function:

- ghci will try to print the result, but a function is not an instance of the `Show` type class.

For example:

```
multTriple 3 4
```

will result in an error:

```
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]
```

Assignment Project Exam Help

9

https://powcoder.com

Add WeChat powcoder

zipWith'

`zipWith'` function can apply operations across two lists.

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : (zipWith' f xs ys)
```

Some of the interesting examples combine infinite lists:

```
zipWith' (*) (replicate 5 2) [1..]

:{
   zipWith' (zipWith' (*))
   [[1,2,3],[4,5,6],[7,8,9]]
   [[9,8,7],[6,5,4],[3,2,1]]
:}
```

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.

Assignment Project Exam Help

11

https://powcoder.com

Add WeChat powcoder

zipWith and flip'

An example of using the flip' function:

```
zipWith (flip' div) [2,2,..] [10,8,6,4,2]
```

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)]
```

Assignment Project Exam Help

13

https://powcoder.com

Add WeChat powcoder

map

However, there is already the map function in Haskell.

Note that the following are equivalent:

```
map (+3) [1,2,3,4,5]
```

```
[x+3 | x <- [1,2,3,4,5]]
```

Using the map function makes code a bit more readable, especially when applying maps of maps.

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
```

Assignment Project Exam Help

15

https://powcoder.com

Add WeChat powcoder

filter Examples

Again, there is a standard library filter function.

Some examples of using the filter function:

```
filter (>3) [1,5,3,2,6,4,6,3,1,2]

filter (==3) [1,2,3,4,5]

filter even [1..10]

let notNull x = not (null x)
in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
```

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]
```

Assignment Project Exam Help

17

https://powcoder.com

Add WeChat powcoder

Simplifying Quicksort

Another nice aspect of `filter` is that we can use it to simplify our `qsort` code:

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) =
 let
 left = filter (<= x) xs
 right = filter (> x) xs
 in
 (qsort left) ++ [x] ++ (qsort right)
```

18

Lazy 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).

Assignment Project Exam Help

19

https://powcoder.com

Add WeChat powcoder

takeWhile

`takeWhile` is a function similar to `filter`:

- returns a list for the first elements of input list that satisfy predicate
- and no elements after one is found to not satisfy predicate.
- see an example of how to parse the first word from a string:

```
takeWhile (/=' ') "elephants know how to party"
```

An example involving quite a few nested functions:

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

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)
:}
```

Assignment Project Exam Help

21

https://powcoder.com

Add WeChat powcoder

But Wait, There's More...

There is also an alternative with an equivalent list comprehension version:

```
sum (takeWhile (<10000) [m^2 | m <- [1..], 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

Assignment Project Exam Help

23

https://powcoder.com

Add WeChat powcoder

Collatz Chain

The recursive function below creates Collatz chains:

```
chain :: Integer -> [Integer]
chain 1 = [1]
chain n
 | even n  = n : chain (n `div` 2)
 | True    = n : chain (3*n + 1)
```

24

Collatz Conjecture

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

Assignment Project Exam Help

25

https://powcoder.com

Add WeChat powcoder

How Many Long Chains?

And we can count chains with more than 15 elements (considering chains generated up to input integer 100):

```
numOfLongChains :: Int
numOfLongChains =
   length (filter isLong (map chain [1..100]))
   where isLong xs = length xs > 15
```

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`

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Lambda Functions

It is often more concise to use anonymous functions known as lambda functions. A simple example first:

```
(\x -> x + 2) 3
```

Then another example using lambda function as parameter:

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15)
(map chain [1..100]))
```

Note that lambdas are expressions, so we can use them anywhere expressions can be used.

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)]
```

Assignment Project Exam Help

29

https://powcoder.com

Add WeChat powcoder

Cases, and Runtime Error

The textbook mentions no cases for lambdas directly, but you can still use `case` constructs:

```
(\(a, b) -> case (a, b) of
        (1, 2) -> 2
        (3, 4) -> 7
        (a, b) -> 1
) (1, 2)
```

The above example applies the lambda function immediately to the pair `(1, 2)`.

Keep in mind that if the match for a pattern fails, a runtime error occurs.

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.

Assignment Project Exam Help

31

https://powcoder.com

Add WeChat powcoder

`flip` Implementation

There are times it is convenient to use `flip`.

But you could use a lambda to emphasize a function is meant to be passed as an argument for:

- `map`, `zipWith`, `filter`, etc.,

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

Note that `flip` reads as if the parameters of the lambda function are not being used until later.

```
map (flip subtract 20) [1, 2, 3, 4]
```

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
```

Assignment Project Exam Help

33

https://powcoder.com

Add WeChat powcoder

Simplify with Currying

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

Notice that the lambda function is not needed because of currying:

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

Also, in general, if you have a function such as

`foo a = bar b a`

you can simplify it to

`foo = bar b`

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.

Assignment Project Exam Help

35

https://powcoder.com

Add WeChat powcoder

Right versus Left

For example:

```
mapr :: (a -> b) -> [a] -> [b]
mapr f xs = foldr (\x acc -> f x : acc) [] xs
```

We could implement the above equivalently with left fold:

```
mapr :: (a -> b) -> [a] -> [b]
mapr f xs = foldl (\acc x -> acc ++ [f x]) [] xs
```

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.

Assignment Project Exam Help

37

https://powcoder.com

Add WeChat powcoder

elem

Yet another implementation of the `elem` function:

```haskell
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldr (\x acc ->
        if x == y then True else acc
    ) False ys
```

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

Assignment Project Exam Help

39

https://powcoder.com

Add WeChat powcoder

Empty List or Not

For your own implementations, you can see the conciseness of `foldl1` and `foldr1` functions.

- choose them when the design of your function does not make sense when given an empty list
- choose `foldl` and `foldr` when it does make sense to process an empty list

40

— Break —

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

`reverse'` and `product'`

We have two implementations of `reverse` function:

```
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []
```

Multiplication will require the correct type class:

```
product' :: (Num a) => [a] -> a
product' = foldl (*) 1
```

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.

Assignment Project Exam Help

43

https://powcoder.com

Add WeChat powcoder

last

Finally, we give another implementation for the `last` function:

```
last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

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)))`.

Assignment Project Exam Help

45

https://powcoder.com

Add WeChat powcoder

Left Fold Nesting Operations

Consider a left fold with `g` as the binary function on the same list:

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

Replace `g` with the `flip (:)` binary function and begin `acc` with an empty list `[]`, the expression becomes:

```
flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6
```

Check an evaluation of the above and we have `[6,5,4,3]`.

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.

Assignment Project Exam Help

47

https://powcoder.com

Add WeChat powcoder

Short Circuiting

Because `&&` short circuits evaluation:

- if the first value is `False`,
- `foldr` on an infinite list with `and'` can short circuit,
- and completes once an element in the list results `False`.
- maybe a dangerous function since all the elements in an infinite list could be `True`

Go ahead and try the expression:

```
and' (repeat False)
```

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

Assignment Project Exam Help

49

https://powcoder.com

Add WeChat powcoder

scan

Try yourself and see the results:

```
scanl (+) 0 [3,5,2,1]
scanr (+) 0 [3,5,2,1]
scanl1 (\acc x -> if x > acc then x else acc)
[3,4,5,3,7,9,2,1]
scanl (flip (:)) [] [3,2,1]
```

The last accumulator value when using `scanl` will be in the last element of the resulting list.

For `scanr`, the first element in the resulting list is the last accumulator value.

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])
```

Assignment Project Exam Help

51

https://powcoder.com

Add WeChat powcoder

Using $ to Apply Functions

```
sqrt 3 + 4 + 9
```

can be equivalently written as

```
(((sqrt 3) + 4) + 9)
```

- the `$` operator applies a function,
  but with the lowest precedence in the expression
- it is described with types as
  `($) :: (a -> b) -> a -> b`
  and defined as
  `f $ x = f x`
- the definition does not make its usefulness explicit

```
sqrt $ 3 + 4 + 9
```

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]
```

Assignment Project Exam Help

53

https://powcoder.com

Add WeChat powcoder

Alternative Syntax

Perhaps you prefer use of `pipe`:

```
:{
 (map (*2) [2..10])
 `pipe`
 (filter (> 10))
 `pipe`
 (sum)
:}
```

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`.

Assignment Project Exam Help

55

https://powcoder.com

Add WeChat powcoder

Function Composition

$$f \cdot g(x) = f\big(g(x)\big)$$

Function composition in Haskell uses dot as operator:

```
(.) :: (b -> c) -> (a -> a) -> a -> c
f . g = \x -> f (g x)
```

```
map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
```

```
map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
```

Note that composition will require exactly one input.

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]]
```

Assignment Project Exam Help

57

https://powcoder.com

Add WeChat powcoder

Currying and Multiline: . and $

We can *make* functions have one parameter with currying:

```
sum (replicate 5 (max 6.7 8.9))
```

```
(sum . (replicate 5) . max 6.7) 8.9
```

We can use multiline with layout syntax after dot operator
(do not use let keyword; this is not a definition; no need to indent):

```
(replicate 2 . product . map (*3) .
(zipWith max [1,2])) [4,5]
```

We cannot do so with $:

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

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.

Assignment Project Exam Help

59

https://powcoder.com

Add WeChat powcoder

Simplifying with . and $

Instead, Haskell has dot notation to help with creating a function in what is termed point-free style:

```
fn x = ceiling (negate (tan (cos (max 50 x))))

fn = ceiling . negate . tan . cos . max 50
```

We cannot just remove `x` on both sides

- because it does not make sense to express `` `cos (max 50)` ``
- that is, we cannot take the cosine of a function.

The second equivalent statement removes notation `x`.

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!

Assignment Project Exam Help

61

https://powcoder.com

Add WeChat powcoder

— Break —

62

— Chapter 6 —

Assignment Project Exam Help

63

https://powcoder.com

Add WeChat powcoder

Modules

A module in Haskell is a file with some type classes, types, and functions defined inside.

- a Haskell program is a collection of modules
- some of the module contents can be exported for other Haskell programs to use
- with more generic code, such as a module, it facilitates 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
```

Assignment Project Exam Help

65

https://powcoder.com

Add WeChat powcoder

Data.List Module Functions

The `Data.List` module allows use of any of its functions.

```
numUnique :: (Eq a) => [a] -> Int
numUnique = length . nub
```

`numUnique` function calls `nub` imported from `Data.List` module to remove duplicate elements from a list.

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)
```

Assignment Project Exam Help

67

https://powcoder.com

Add WeChat powcoder

Naming Conflicts

A naming conflict with existing functions needs the following to allow import…

```
import qualified Data.Map
```

…but then every function from the module we call must be prefixed with `Data.Map.`

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

prefix with `M`

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 ` . `

Assignment Project Exam Help

69

https://powcoder.com

Add WeChat powcoder

Working with Text

```
text = words "just remember that the things you put into
your head are there forever he said you might want to
think about that you forget some things dont you yes you
forget what you want to remember and you remember what you
want to forget"
```

The above paragraph is from a fictional novel named The Road (a future dystopic survival story).

```
n = length text

group text

group . sort $ 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
```

Assignment Project Exam Help

71

https://powcoder.com

Add WeChat powcoder

Frequency Analysis

Note that all the frequencies added up for the paragraph should equal `100` percent:

```
let
wordFreqs =
    map (\ws ->
       ( head ws,
         (fromIntegral (length ws) / (fromIntegral n))*100
       )
    ) . group . sort $ text

foldl (\acc x -> acc + snd x) 0 wordFreqs
```

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

Assignment Project Exam Help

73

https://powcoder.com

Add WeChat powcoder

`Data.List` module also has function `isInfixOf` (that does the same thing as the `isIn` function).

```
import Data.List

let
isIn :: (Eq a) => [a] -> [a] -> Bool
sublist `isIn` xs = any (sublist `isPrefixOf`) (tails xs)
```

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
```

Assignment Project Exam Help

75

https://powcoder.com

Add WeChat powcoder

- keep the letters within lowercase characters
- leave spaces alone (but easier to hack original message)

```
caesar offset msg = map (\c -> if c /= ' ' then chr $ 97 +
(ord c - 97 + offset + 26) `mod` 26 else ' ') msg
```

A decrypt function is easy if we already have encryption:

```
decode :: Int -> String -> String
decode offset msg = caesar (negate offset) msg
```

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
```

Assignment Project Exam Help

77

https://powcoder.com

Add WeChat powcoder

Avoiding Deferral

```
((0 + 1) + 2) + 3
```

See how the elements are deferred into nested sums:

- we should be able to avoid the deferral if we want.
- the `Data.List` module has such a version of the `foldl` function named `foldl'`

The order of execution and operations happen something as in the following:

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

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.

Assignment Project Exam Help

79

https://powcoder.com

Add WeChat powcoder

find

- check out the type for the function `:t find`

```
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

  - do not worry about `Foldable` type class for now
  - notice `Maybe a` type takes on two possible values
    - the `Nothing` type
    - `Just a` with whatever the type of `a` could be
  - this allows the `find` function to return either:
    - some value of type `a`
    - or fail
- the point of `find` is to return the first value in a list that makes satisfies a predicate passed in

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
```

Assignment Project Exam Help

81

https://powcoder.com

Add WeChat powcoder

Get What We Want Quickly

Now try everything together:

```
firstTo40 :: Maybe Int
firstTo40 = find (\x -> digitSum x == 40) [1..]
```

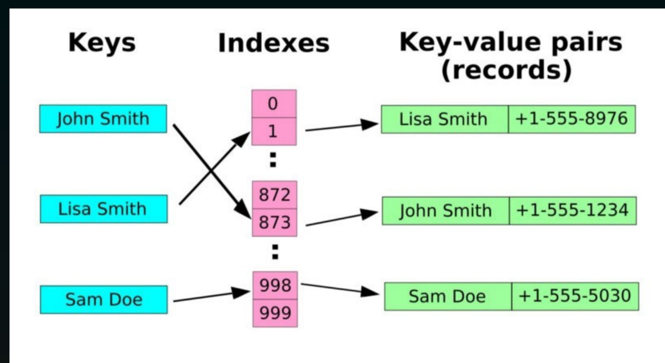
Edit slightly to allow search for arbitrary input:

```
firstTo :: Int -> Maybe Int
firstTo n = find (\x -> digitSum x == n) [1..]
```

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

Assignment Project Exam Help

83

https://powcoder.com

Add WeChat powcoder

Association List:

Word Keys and Phone Number Data

One example of a map is a list of people's name associated with their phone number:

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

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
    )
```

Assignment Project Exam Help

85

https://powcoder.com

Add WeChat powcoder

Alternative to Get Data

Get data using `find` or `fromJust` functions:

```
let

findKey :: (Eq k) => k -> [(k, v)] -> v
findKey key xs = snd . head .
    filter (\(k, v) -> k == key) $ xs
```

this will crash on error if there is no pair with matching key

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
```

Assignment Project Exam Help

87

https://powcoder.com

Add WeChat powcoder

Context of List vs Elements

The previous example used `foldr` instead of recursing through a list, which is much easier to read.

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`

Assignment Project Exam Help

89

https://powcoder.com

Add WeChat powcoder

Maps vs Association Lists

Consider the type `:t Map.fromList`:

```
Map.fromList :: Ord k => [(k, a)] -> Map.Map k a
```

- the keys are orderable, so that the data can be arranged and accessed more efficiently
- this was not the case with association lists on their own

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")
 ]
```

Assignment Project Exam Help

91

https://powcoder.com

Add WeChat powcoder

Create New Maps

Haskell maintains immutable data:

```
Map.lookup "betty" phoneBook
Map.lookup "wendy" phoneBook
Map.lookup "grace" phoneBook
```

Create a new map from an old one

- insert a new key and value pair with `Map.insert` function:

```
Map.lookup "grace" phoneBook
let newBook = Map.insert "grace" "341-9021" phoneBook
Map.lookup "grace" newBook
```

(:t Map.insert)

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.

Assignment Project Exam Help

93

https://powcoder.com

Add WeChat powcoder

Convert Map Types

We can convert phone numbers in our map to an Int list:

- first, create a function to convert for one phone number
  - use `digitToInt` from `Data.Char` module
  - use `filter` and `isDigit` together to get rid of any dashes

```
string2digits :: String -> [Int]
string2digits = map digitToInt . filter isDigit

string2digits . fromJust $ Map.lookup "grace" newBook
```

Now `Map.map` can convert with `string2digits` to our collection of phone numbers as [Int]:

```
let intBook = Map.map string2digits phoneBook
```

give `Map.lookup` a try.

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)

Assignment Project Exam Help

95

https://powcoder.com

Add WeChat powcoder

Repeated Keys with More Data

```
let
phoneBook =
 [("betty", "555-2938")
 ,("betty", "342-2492")
 ,("bonnie", "452-2928")
 ,("patsy", "493-2928")
 ,("patsy", "943-2929")
 ,("patsy", "827-9162")
 ,("lucille", "205-2928")
 ,("wendy", "939-8282")
 ,("penny", "853-2492")
 ,("penny", "555-2111")
 ]
```

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
```

Assignment Project Exam Help

97

https://powcoder.com

Add WeChat powcoder

Querying Maps

Observe how `max` can be used to result in only the one pair we might want for each unique key:

```
Map.fromListWith max
[(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
```

Give this a try with the `(+)` function instead to add the values for each unique key.

Such an operation would be similar to an SQL group query with sum applied to each group.

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
```

Assignment Project Exam Help

99

https://powcoder.com

Add WeChat powcoder

Thank You!

Questions?

100