Week 6

# Ch 9: More Input and More Output
# Ch 10: Functionally Solving Problems

University of the Fraser Valley

Dr. Russell Campbell

Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming



1



2

# Functor Operations Example
(David Semke)

```
import Data.List

-- Get a list of unique sums from the sums of all possible pairs
of items from list1 and list2

list1 = [5, 6, 7, 8]
list2 = [1, 2, 3, 4]

method1 = nub [a+b | a <- list1, b <- list2]

method2 = nub $ fmap (+) list1 <*> list2

method3 = nub $ (+) <$> list1 <*> list2

method4 = nub $ pure (+) <*> list1 <*> list2

method5 = nub $ [(+)] <*> list1 <*> list2
```



3



# Files and Streams

4

## Reading File Input

Save the following text data as `haiku.txt` in a subfolder named `data`:

```
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
```

Then save the following script as `capslocker.hs`:

```
{- ### capslocker.hs v1 ### -}
import Control.Monad
import Data.Char

main = forever $ do
    l <- getLine
    putStrLn $ map toUpper l
```



5



## Command-Line Compile and Execute

6

## getContents (1)

We can simplify the `capslocker` program by making use of `getContents` function.

- `getContents` does lazy I/O in Haskell
  - it gets the input when required later, instead of right away
  - it reads standard input until it gets an end-of-file character
  - its type is `getContents :: IO String`
  - it takes care of how much input to get for us, unlike `forever`

```
{- ### capslocker.hs v2 ### -}
import Data.Char

main = do
    contents <- getContents
    putStr $ map toUpper contents
```



7



capslocker

## getContents (2)

getContents

getContents

getContents

8

# *lines and unlines

We can practice lazy input like this more with a program that restricts which input should be displayed depending on length:

```
{- ### shortLinesOnly.hs v1 ### -}
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly =
        unlines . filter (\line -> length line < 10) . lines
```

- `lines` and `unlines` functions behave like `words` and `unwords`
- but for input separated by end-of-line characters split into elements of a list and back



9



# *Execute shortLinesOnly

10

## * `shortLinesOnly` version 2

The previous implementation has a useful function to do the same kind of operations on each line of input:

```
{- ### shortLinesOnly.hs v2 ### -}
main = interact shortLinesOnly


shortLinesOnly :: String -> String
shortLinesOnly =
        unlines . filter (\line -> length line < 10) . lines
```

- `` `interact` `` takes a function of type `` `String -> String` `` as a parameter and gives back an I/O action
  - the I/O action performs the same execution as the previous version of our `` `shortLinesOnly` `` program
  - i.e.: it processes one line at a time of streamed input
- * we can either redirect, or type lines of input ourselves



11



## *Palindromes (1)

palindrome

12

*Palindromes (2)

```
{- ### palindromes.hs ### -}
main = interact respondPalindromes

respondPalindromes :: String -> String
respondPalindromes =
    unlines
    . map (\line ->
            if isPal line
                then "palindrome"
                else "not palindrome“
          )
    . lines

isPal :: String -> Bool
isPal xs = xs == (reverse xs)
```



13



*Palindromes (3)

14

— Reading and Writing Files —



15





openFile

and
hGetContents

16

## openFile (1)

- `openFile :: FilePath -> IOMode -> IO Handle`
  - parameters take:
    - (1) a specified file,
    - (2) an IOMode (`ReadMode`), and
    - (3) an I/O action to open a file and yield the file's associated handle
- `FilePath` is just a type synonym for `String`, i.e.: `type FilePath = String`
- `IOMode` is defined like so:
  - `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode`
  - (note that it is not `IO Mode` with a space, which would be an I/O action with yield of `Mode`)
  - `IOMode` is just an enumeration



17



## openFile (2)

getContents
hGetContents

hGetContents

putStr

hClose

18

## withFile (1)

Function `withFile` compacts the execution of our previous program, and it has the following signature:

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
```

- `FilePath` takes a specified file
- `(Handle -> IO a)` function
- lastly, `withFile` returns an I/O action
  - it will open the file
  - perform some action with the function passed in
  - close the file
  - if anything goes wrong, `withFile` makes sure to close the file



19



## withFile (2)

withFile

withFile

20

## Exceptions

Errors during execution that happen (e.g.: calling `head` on an empty list) typically print an error message:

- these are called exceptions
- `withFile` ensured a file handle is closed when an exception is raised
- for any resource (such as a file), we need to ensure it is released given any situation or error

Then the common design for working with a resource and dealing with input and output using it is as follows:

- get the resource as an I/O action
- ensure it can be closed regardless of exception
- process the data of the resource as an I/O action
- (altogether, these describe an I/O action of the resource type)



21



## — The Bracket Function —

22

## The `bracket` Function

Then the following type for the `bracket` function…

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO a
```

…matches its parameters (and return type) for the common design described.

- the first `IO a` is the resource obtained, but as an I/O action
- `(a -> IO b)` is a function to execute regardless of exception
- `(a -> IO c)` is a function to process the resource
- the last `IO a` needs to match context of input I/O action `IO a`



23



## Using the `bracket` Function

bracket

24

— Working with Handles —



25



Functions
for
Handles

26

# Files as Large Strings

Using files as if they were large strings is very common, so we have three more functions to help with this:

- `` `readFile :: FilePath -> IO String` `` take a specified file and performs I/O read action (lazily) and can bind to some variable
  - a bit more concise than calling `` `openFile` `` and then `` `hGetContents` ``
- `` `writeFile :: FilePath -> String -> IO ()` `` takes a specified file, a string, and performs I/O write action
  - if the file already exists, its contents are first erased before writing
- `` `appendFile :: FilePath -> String -> IO ()` ``
  - same as `` `writeFile` ``, but does not erase contents first, and instead appends the given string after the contents of the file



27



# File Reading and Writing

(Simranjit Singh)

28

`appendFile`
(Hunter Klassen)

```
import System.IO

main = do
    student <- getLine
    appendFile "classList.txt" (student ++ ", ")
```

File `classList.txt`:

```
Klassen,
```



29



Directories and Files
(Amy Campbell)

30

Directories and Files Functions
(Amy Campbell)

- getDirectoryContents will show `.` and `..`
- listDirectory is the same, but does not show `.` and `..`
- (we have seen appendFile in a previous example)



31



— To-Do List —

32

## To-Do List (1)

We have enough to write an application to keep a list of things we would like to do and store them in a file for us.

Creating files involves POSIX libraries, so for now, create the `data/todo.txt` file, and write the following program:

```
{- ### todo.hs v1 ### -}
import System.IO

main = do
    todoItem <- getLine
    appendFile "data/todo.txt" (todoItem ++ "\n")
```



33



## To-Do List (2)

34

## Deleting To-Do Items (1)

```
{- ### deletetodo.hs v1 ### -}
import System.IO
import System.Directory
import Data.List

main = do
    conetents <- readFile "data/todo.txt"
    let
        todoTasks = lines contents
        numberedTasks =
            zipWith (\n line -> show n ++ " - " ++ line)
                [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    mapM_ putStrLn numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
```

(cont'd on next slide...)



35



## Deleting To-Do Items (2)

36

## Deleting To-Do Items (3)

- `todoTasks` stores a list of lines from the `todo.txt` file
  - it looks something like

    `["iron…","dust…","take the salad…"]`
- `zipWith` prefixes a number to each line in the `todoTasks` list
  - `numberedTasks` looks something like

    `["0 – iron…","1 – dust…","2 - take the salad…"]`
- `mapM_` prints out the `numberedTasks`, each element on a separate line
- `number` gets assigned to a choice from the user of which numbered line to delete



37



## Deleting To-Do Items (4)

!! delete

unlines

openTempFile

openTempFile

"."

"temp"

38

## More Exceptions

There are some issues with the `deletetodo.hs`:

- if the program terminates from some kind of error
  - the temporary file is created,
  - but not renamed
- we need functionality similar to `bracket` where
  - a resource gets cleaned up automatically when we are done
  - but only if there is some kind of exception



39



## *Deleting To-Do Items:

Version 2

40

# *Deleting To-Do Items:
## Version 2

The parameters of `bracketOnError`:

- the resource first
- what to do on an exception
- what to do as expected normally

```
let
    number = read numberString
    newTodoItems =
        unlines $ delete (todoTasks !! number) todoTasks
bracketOnError (openTempFile "." "temp")
    (\(tempName, tempHandle) -> do
        hClose tempHandle
        removeFile tempName
    )
    (\(tempName, tempHandle) -> do
        hPutStr tempHandle newTodoItems
        hClose tempHandle
        removeFile "data/todo.txt"
        renameFile tempName "data/todo.txt"
    )
```



41



— Command-Line Arguments —

42

## Command-Line Arguments

The To-Do App limitations only work with the one text file, and only add or delete one item from a list one at a time.

Instead, we can specify what we would like our program to do exactly when we execute it. We use `System.Environment`:

- it lets us read command-line arguments:

```
{- ### argTest.hs ### -}
import System.Environment
import Data.List

main = do
    args <- getArgs
    progName <- getProgName
    putStrLn "The arguments are:"
    mapM putStrLn args
    putStrLn "The program name is:"
    putStrLn progName
```



43



## Combine To-Do App Functions

44

## User Enters Command

The program start will decide between which function to run which matches the first command argument input:

```
{- ### todo.hs part 1 ### -}
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove

main = do
    (command:argList) <- getArgs
    dispatch command argList
```



45



## Example User Execution

main

command
dispatch

main add

46

## add

Next, we implement the three functions for our app, starting with `add`:

```
{- ### todo.hs part 2, add v1 ### -}
add :: [String] -> IO ()
add (filename:toDoItems) =
    appendFile filename (unwords (toDoItems ++ ["\n"]))
```

- (no errors dealt with for mismatched file names until later)
- the rest of the list is dealt with as one long task to be appended on a new line at the end of the user-specified file
  - `unwords` will concatenate all string elements in the list together as one string separated by a space



47



## Foldable and Exception Modules

48

## View To-Do List

`Foldable` module is for `` `mapM_` `` and `Exception` module is for use of `` `bracketOnError` `` to handle potential file issues.

```
{- ### todo.hs part 4 ### -}
view :: [String] -> IO ()
view (filename:remainder) = do
    contents <- readFile filename
    let
        todoTasks = lines contents
        numberedTasks =
            zipWith (\n line -> show n ++ " - " ++ line)
                [0..] todoTasks
    putStrLn "These are your To-Do items:"
    mapM_ putStrLn numberedTasks
```



49



## Remove from To-Do List (1)

```
remove
remove

    mapM_
```

50

## Remove from To-Do List (2)

```
let
    number = read numString
    newTodoItems =
            unlines $ delete (todoTasks !! number) todoTasks
bracketOnError (openTempFile "." "temp")
    (\(tempName, tempHandle) -> do
        hClose tempHandle
        removeFile tempName
    )
    (\(tempName, tempHandle) -> do
        hPutStr tempHandle newTodoItems
        hClose tempHandle
        removeFile "data/todo.txt"
        renameFile tempName "data/todo.txt"
    )
```



51



## Example To-Do List

view

dispatch

bump

52

— Handling Bad Input —



53



doesntExist
doesntExist

doesntExist

add

54

— Randomness —



55



random

stack

The `random` Function

random

56

## Random Generators

- `RandomGen` type class is for types as sources of randomness
- `Random` type class is for type values resulting from random source

Then interpreting the type for the `random` function...

- input is a random generator (or source of randomness)
- output is a pair: random value and another random generator

The pair facilitates a sequence of random values, controlled by generators.



57



## Generating a Random Number

mkStdGen

mkStdGen

StdGen

58

## Specific Types of Random Value

We sometimes need to make sure we have the random number of a concrete type; specify with type annotation:

```
random (mkStdGen 100) :: (Int, StdGen)
```

- notice that it gives the exact same random number in the first element of the pair
- we can change the first element depending on which random generator is passed in

```
random (mkStdGen 101) :: (Int, StdGen)
(5022189999818777644,StdGen
{unStdGen = SMGen 1869071494976701883 15060681878671775511})
```



59



## Other Random Value Types

```
Float
```

```
Bool
```

60

— Toss a Coin —



61



Toss Three Coins

threeCoins

threeCoins

62

*Probabilities

Could you tally and calculate sample probabilities for the results?

- how often is the first coin `False` vs `True` after 10 executions?
- what about the other coins?
- were any triples repeated?
- with many executions,
  does the result get closer to 50% probability for each coin?
- is the random generator a fair one?
- there are quite a few things we could analyze mathematically



63



— Sequences of Random Numbers —

64

## *randoms

We want to avoid manually passing in generators to the `random` function to work with some sequence.

- thankfully, there is a `randoms` function
  - the input is a random generator
  - the output is a sequence of random numbers

For example:

```
take 5 $ randoms (mkStdGen 100) :: [Int]
```

- notice that a random generator is not returned.
  - this is due to the recursive design of `randoms`
  - we will implement our own version, so you can see



65



## *Integer Random Sequence

66

## *Limiting Recursion (1)

If we want to get back a generator, we need to customize a function to return a finite number of random values:

```
finiteRandoms :: (Num n, Eq n) => n -> StdGen -> ([Int], StdGen)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
    let
        (value, newGen) = (random gen) :: (Int, StdGen)
        (restOfList, finalGen) = finiteRandoms (n-1) newGen
    in (value:restOfList, finalGen)
```



67



## *Limiting Recursion (2)

newGen

Int

StdGen

68

# *randomR

We may also want to control the range of possible values we would like as a result for a random number.

- this is provided by function `randomR`

```
randomR :: (RandomGen g, Random a) => (a, a) -> g -> (a, g)
```

- it works pretty much the same as `random`, but it takes a pair of elements describing a range of values
  - e.g.: `randomR (1,6) (mkStdGen 359353)`
  - the above would give a value between `1` and `6`, inclusive

```
take 10 $ randomRs ('a', 'z') (mkStdGen 100)
```



69



— Randomness and I/O —

70

## getStdGen

Randomness is not helpful if we execute a program on the command line and it always gives the same random values.

- so far, this would be all we could program
- however, there is a `getStdGen` I/O action that will set up a different *global* generator for us when it is bound during an I/O action
- we can then use the global generator as we have with generators like the `StdGen` type we have already used

```
{- ### randomString.hs ### -}
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a', 'z') gen)
```



71



## Different Result Each Execution

getStdGen

newStdGen

72

# *Guessing Game (1)

- try swapping in `getStdGen` for `newStdGen` to see repeated string
- however, if you use `getStdGen` to bind again after `newStdGen` binds, we will get yet a third different global generator

We can write a more substantial program to play a game with the user to guess a randomly generated number:

```
{- ### guessGame.hs ### -}
import System.Random
import Control.Monad(when)

main = do
    gen <- getStrGen
    askForNumber gen
```



73



# *Guessing Game (2)

askForNumber
askForNumber
randomR
askForNumber

74

# *Exit Guessing Game

- you can exit the program by not typing a guess and press `ENTER`
  - (the `when` condition is `False`)
- `when` is used for the situation that an `else` clause is not wanted
- look up the function `reads` and use it to handle bad user input, such as "haha"
  - it returns an empty list upon failure to read a string
  - else, the return fits the pattern `[(input string, remainder of input)]`

Note that the guessing game program is recursive.

- each time the user guesses, `askForNumber` is called another recursive level



75



— Bytestrings —

76

## Thunks

We can treat files, lists, and even infinite lists as streams to access them only when a program absolutely needs to.

- `[1,2,3,4]` is just presentation for `1:2:3:4:[]`
- i.e. the right expression is a *promise* of a list (it is not evaluated to one yet)
- such expressions that are deferred computations are called thunks
- this means we can work with collections of infinite elements
- obviously, this behaviour is not likely to be efficient
  - most of the time the extra work by a computer does not get in our way
  - it does become a problem for big files, including their manipulation



77



## Bytestrings

bytestrings

8

78

— Strict and Lazy Bytestrings —



79



Levels of Laziness

80

## Chunk or Thunk

For strict bytestrings:

- we cannot have an infinite number of byte elements
- evaluating one element means evaluating all others
  - i.e.: there is ***no*** laziness, and no thunks involved at all

For lazy bytestrings:

- each element is a thunk, so they are slow for some uses
- elements are stored in chunks (**not** thunks) of 64 KB each
  - evaluating a byte in a chunk means the whole chunk will be evaluated
  - processing a file with a lazy bytestring will be done chunk by chunk
  - this way, memory usage will not suffer
  - 64 KB likely fits within the CPU L2 cache



81



## Lazy versus Strict Bytestrings

ByteString

Word8

B

S

82

# Data within Bytestrings

Then `B` refers to the lazy bytestring module and `S` the strict module

- (we mostly will show use of the lazy version of functions)

The first function `pack :: [Word8] -> ByteString` converts:

- a list (very lazy) to a less lazy form (only at 64 KB chunks)
- `Word8` is similar to `Int`, but restricted to values from `0` to `255`
  - also categorized within the `Num` type class
  - e.g.: literal `5` is polymorphic, and can act like a `Word8` type value



83



# Simple Examples of Lazy Bytestrings

pack

Word8

pack

84

## Conversion of Data

If we want access to the elements of the lazy bytestring, we must first unpack it into a list of `Word8` elements:

```
let by = B.pack [98,111,114,116]
B.unpack by
```

There are functions such as `fromChunks` to convert a list of strict bytestrings into a one lazy bytestring:

```
B.fromChunks [S.pack [40..42], S.pack [43..45], S.pack [46..48]]
```

- the result looks like a string
- `fromChunks` converts a list of strict bytestrings to a lazy bytestring
- the inverse operation `toChunks` can convert a lazy bytestring into a list of strict ones



85



## Prefix Like a List

cons

86

## Other Modules and Functions

Conversion between lazy bytestrings, strings (lists of chars), and strict bytestings has other modules you may want:

- the data type for Word8 is found in Data.Word (but not that many functions)
- Data.Text has chunksOf function to split text into smaller pieces (use with Data.Text.pack to convert strings to Text type)
- Data.ByteString.Builder has functions to help with efficient processing of text data (example in builder.hs)



87



## Files and `ByteString`

88

— Copying Files with Bytestrings —



89



Example of Reading a File (1)

90

## Example of Reading a File (2)

```
copy source dest = do
    contents <- B.readFile source
    bracketOnError
        (openTempFile "." "temp")
        (\(tempName, tempHandle) -> do
            hClose tempHandle
            removeFile tempName
        )
        (\(tempName, tempHandle) -> do
            B.hPutStr tempHandle contents
            hClose tempHandle
            renameFile tempName dest
        )
```



91



## Notes on Processing Files

92

## When Processing Files

Ultimately, for a program to switch from normal file processing to a bytestring version would be to:

- make the necessary imports
- prefix the read and writes with corresponding module for desired bytestring type

Consider switching for better performance on larger files if you are noticing slowdown.

NLP will likely require processing of very large text corpus.



93



— Chapter 10 —

94

## Substantial Code Examples

This chapter is good practice to help you remember the functions and syntax of Haskell so far.



95



— Reverse Polish Notation Calculator —

96

## Infix Notation

Infix notation for writing the common algebraic operations as functions:

- e.g.: `3 + (6 * 2)`

We have seen how to write them using infix with backticks:

- for example: `` `elem` ``



97



## Reverse Polish Notation

98

- let's step through to the result of the expression

  `10 4 3 + 2 * -`

- the result should be `-4`



99



— Writing an RPN Function —

100

## Stack Processing

We want to parse a string and return a numeric result, so a function that evaluates an RPN expression is described as:

```
solveRPN :: String -> Double
```

- should have a space between values and operators
- parse so each token in expression will be an element of a list
  - e.g.: `["10","4","3","+","2","*","-"]`
- process expression element by element, left to right, with `foldl`
  - use accumulator as our stack, so result of the `foldl` should be a stack
- how do we want to represent a stack?
  - if we use a list, its head could be the top of a stack
  - the front of a list is much faster to add an element than at the back



101



## RPN Calculator Design

102

## RPN Calculator Code

Now let's implement the `foldingFunction` that:

- is applied to each element of a list via `foldl`
- so we must work with accumulator as its first parameter (our stack)
- an element from the top of the stack as its second parameter (either an operation or a number)
  - remember, this element will be the top of the stack

```
solveRPN :: String -> Double
solveRPN =
    head . foldl foldingFunction [] . words
        where
        foldingFunction (x:y:ys) "+" = (y + x):ys
        foldingFunction (x:y:ys) "*" = (y * x):ys
        foldingFunction (x:y:ys) "-" = (y - x):ys
        foldingFunction (x:y:ys) "/" = (y / x):ys
        foldingFunction xs numberString = read numberString:xs
```



103



## Example of Using RPN

104

— Adding More Operations —



105



More Operators

106

## Example with More Operations

- the operators do not even need to be binary
- even `sum` can just pop all numbers off the stack and push back the single result

```
solveRPN "2.7 ln"
0.9932517730102834
solveRPN "10 10 10 10 sum 4 /"
10.0
solveRPN "10 10 10 10 10 sum 4 /"
12.5
solveRPN "10 2 ^"
100.0
```

There is no error handling with this calculator, but as it is, having such concise code with Haskell is evident in this application.



107

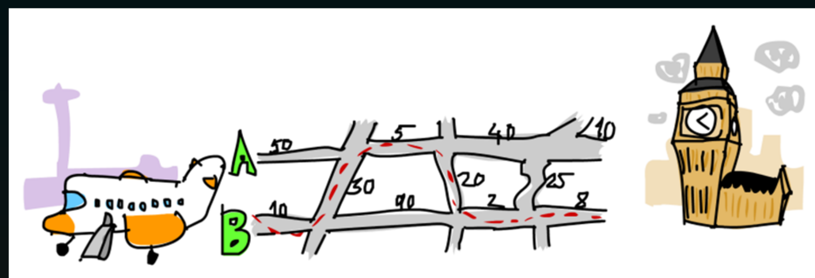


— Heathrow to London —

108

# Solving Problems with Data Types

Observe the road system with times (in minutes) for travelling each road toward the famous "Big Ben" clock tower in London:






109



# Data Types

110

## Section and RoadSystem

Design a data structure describing three values at a time:

```
data Section = Section {
    getA :: Int
    , getB :: Int
    , getC :: Int
} deriving (Show)

type RoadSystem = [Section]
```

- `Section` record stores three `Int` values for road sections A, B, and C
- `RoadSystem` list has elements of type `Section`



111



## Road Data, Label, and Path Type

112

## Toward a Solution

A final function to describe a solution:

```
optimalPath :: RoadSystem -> Path
```

A solution with `roadsToTower` as input should evaluate to:

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

Each step we want optimal path to either intersection along A or B, in parallel, to then move on to the next step.

- we build up the best route, intersection by intersection, in parallel
- as a function, this would look like `roadStep :: (Path, Path) -> Section -> (Path, Path)`
- we are keeping track of two paths



113



## roadStep

114

## `roadStep` Notes

- `` `sum (map snd pathA)` `` totals times for the roads chosen in a path
- note that `` `A` `` is just enum label, while `` `a` `` is a variable of type `` `Int` ``
- forming paths as we step will prefix each new pair, instead of concatenation at the end
  - `` `:` `` operation at start of list is faster than `` `++` `` at end of list
  - the paths are unfortunately backward order, but we can reverse the result easily



115



## Using `roadStep`

116

## `foldl` to a Solution

Use the `roadStep` function to write the `optimalPath` solution, by repeatedly calling `roadStep` per `Section`:

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
    let (bestAPath, bestBPath) = foldl roadStep ([],[]) roadSystem
    in if sum (map snd bestAPath) <= sum (map snd bestBPath)
        then reverse bestAPath
        else reverse bestBPath
```

Give the solution a try:

```
optimalPath roadsToTower
```

- the last pair of the path `(C, 0)` can be ignored



117



— Using Input for the Road System —

118

## groupsOf

Design a few utility functions to help with parsing standard input for setting up the `RoadSystem` data.

The following will help split a list into equal-sized sections:

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

for input `[1..10]`, `groupsOf 3` should evaluate to:

```
groupsOf 3 [1..10]
[[1,2,3],[4,5,6],[7,8,9],[10]]
```



119



## Path Solution

120

## Road Data

Save the following in `paths.txt` for the `Section` times:

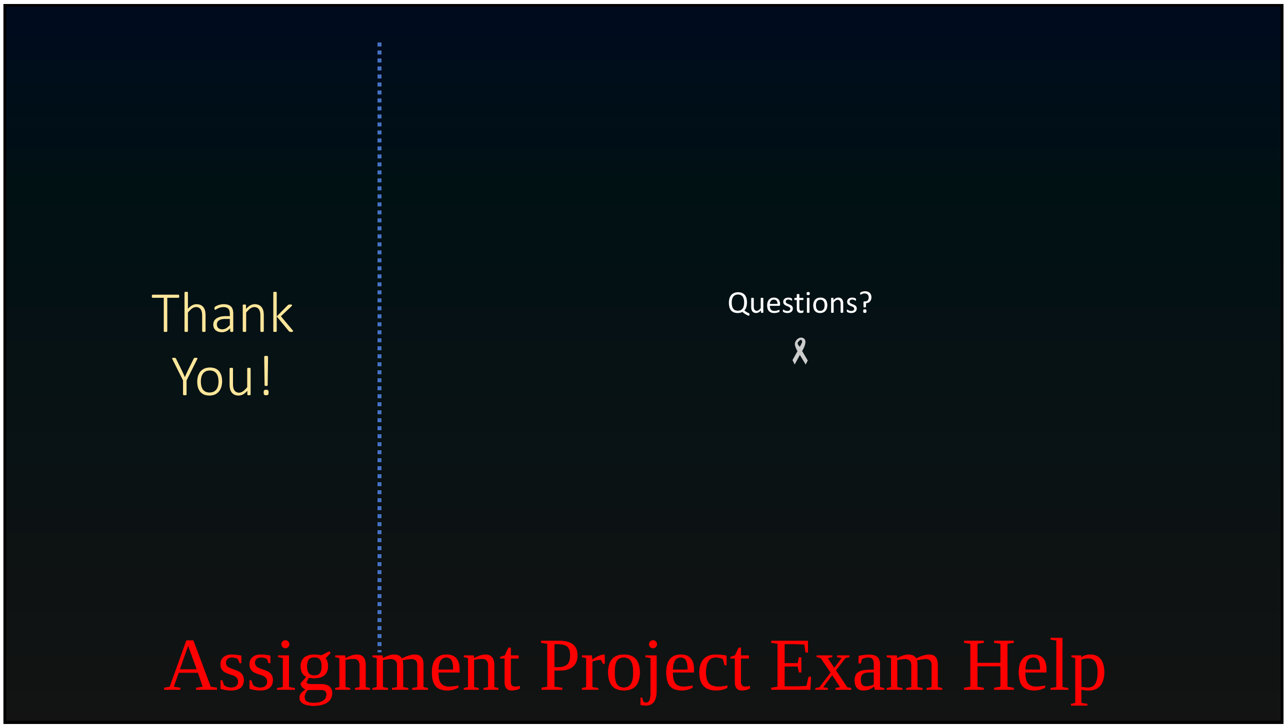
```
50
10
30
5
90
20
40
2
25
10
8
0
```



121

## Finishing Up

122




123