CSC 372, Spring 2016
Assignment 4
Due: Friday, February 26 at 23:59:59

ASSIGNMENT-WIDE RESTRICTIONS

There are three assignment-wide restrictions:

1. Minus some exceptions that are noted for `group.hs` and `avg.hs`, the only module you may import is `Data.Char`. The purpose of this restriction is so that students don't waste time scouring dozens of Haskell packages in search of something that might be useful. `Data.Char` and the `Prelude` have all you need!

2. List comprehensions may **not** be used. They are interesting and powerful but due to time constraints we don't cover them. I want your attention focused on the elements of Haskell that we have covered.

3. Recall the idea put forth on slide 268: To build your skills with higher-order functions I want you to solve most of these problems while pretending that you don't understand recursion! Specifically, **except for problem 1, `warmup.hs`, you may not WRITE any recursive code!** Instead, use higher-order functions like `map`, `filter`, and the various folds. Those functions, and other Prelude functions, might themselves be recursive but that's no problem—it's OK to use recursive functions but you're prohibited from writing any recursive functions.

Make an `a4` symlink

Just like you did for `a3`, make an `a4` symlink:

```
$ cd ~/372/a4
$ ln -s /cs/www/classes/cs372/spring16/a4
```

Use the tester!

Just like for assignment 3, there's a tester for this assignment. **Don't just "eyeball" your output—use the tester!** I won't have any sympathy for those who fail tests during grading simply because they didn't want to bother with the tester! We'll be happy to help you with using the tester and understanding its output.

The tester is in `a4/tester`. Run it with "`a4/tester` *`PROBLEM-NAME`*". To maybe save a little typing, `a4/t` is symlinked to `a4/tester`, so you can run the tester with just `a4/t`.

Problem 1. (7 points) `warmup.hs`

This problem is like `warmup.hs` on assignment 3—I'd like you to write your own version of some functions from the Prelude: `map`, `filter`, `foldl`, `foldr`, `any`, `all`, and `zipWith`.

The code for `map`, `filter`, and `foldl` is in the slides and the others are easy to find, but I'd like you to start with a blank piece of paper and try to write them from scratch. If you have trouble, go ahead and look for the code. Study it but then put it away and try to write the function from scratch. Repeat as needed.

To avoid conflicts with the Prelude functions of the same name, use these names for your versions:

Your function	Prelude function
`mp`	`map`
`filt`	`filter`
`fl`	`foldl`
`fr`	`foldr`
`myany`	`any`
`myall`	`all`
`zw`	`zipWith`

You should be able to write these functions using only pattern matching, comparisons in guards, list literals, cons (`:`), subtraction, and recursive calls to the function itself. Experiment with the Prelude functions to see how they work.

You might find `foldl` and `foldr` to be tough. Don't get stuck on them!

Just like for `a3`'s `warmup.hs`, you can use a `-t` option with the tester to name a specific function to test. Example: "`a4/tester warmup -t mp`". Note that `-t mp` follows `warmup`.

This problem, `warmup.hs`, is the only problem on this assignment in which you can write recursive functions.

Problem 2. (3 points) `dezip.hs`

Write a function `dezip list` that separates a list of 2-tuples into two lists. The first list holds the first element of each tuple. The second list holds the second element of each tuple. **Don't overlook the additional restrictions for this problem following the examples.**

```
> :t dezip
dezip :: [(a, b)] -> ([a], [b])

> dezip [(1,10),(2,20),(3,30)]
([1,2,3],[10,20,30])

> dezip [("just","testing")]
(["just"],["testing"])

> dezip []
([],[])
```

ADDITIONAL RESTRICTIONS for `dezip.hs`:

The only functions your solution may use are `map`, `fst` , (`.`) (i.e., composition), and this function:

```
swap (x,y) = (y,x)    -- include this line in your solution
```

Remember that writing recursive functions is prohibited.

This function is just like the Prelude function `unzip`, but with a hopefully different enough name that you don't test with `unzip` by mistake!

Problem 3. (2 points) `repl.hs`

Write a function `repl` that works just like `replicate` in the Prelude.

```
> :t repl
repl :: Int -> a -> [a]

> repl 3 7
[7,7,7]

> repl 5 'a'
"aaaaa"

> repl 2 it
["aaaaa","aaaaa"]
```

ADDITIONAL RESTRICTION for `repl.hs`: You may not use `replicate`.

Remember the assignment-wide prohibition on recursion.

This is an easy problem; there are several ways to solve it using various functions from the Prelude. Just for fun you might see how many distinct solutions you can come up with.

Problem 4. (2 points) `doubler.hs`

Create a function named `doubler` that duplicates each value in a list:

```
> :t doubler
doubler :: [a] -> [a]

> doubler [1..5]
[1,1,2,2,3,3,4,4,5,5]

> doubler "bet"
"bbeett"

> doubler [[]]
[[],[]]

> doubler []
[]
```

RESTRICTION: Your solution must look like this:

```
doubler = foldr ...
```

That is, you are to bind `doubler` to a partial application of `foldr`. Think about using an anonymous function.

Replace the `...` with code that makes it work. Thirty characters should be about enough but it can be as long as you need.

Problem 5. (2 points) `revwords.hs`

Using point-free style (slides 301-303), create a function `revwords` that reverses the "words" in a string. Assume the string contains only letters and spaces.

```
> revwords "Reverse the words in this sentence"
"esreveR eht sdrow ni siht ecnetnes"

> revwords "testing"
"gnitset"

> revwords ""
""
```

You'll want to use the `words` function on this one.

Problem 6. (4 points) `cpfx.hs`

Once the `a3` deadline has passed, `a4/whm_cpfx.hs` will have my solution for `cpfx` from assignment 3. (Remind me if I forget!) The `cpfx` function is recursive. Rewrite that function to be non-recursive but still make use of my `cpfx'` function, the code for which is to be a part of your solution. Yes, `cpfx'` is recursive. That's OK.

See the assignment 3 write-up for examples of using `cpfx`.

Problem 7. (7 points) `nnn.hs`

The behavior of the function you are to write for this problem is best shown with an example:

```
> nnn [3,1,5]
["3-3-3","1","5-5-5-5-5"]
```

The first element is a `3` and that causes a corresponding value in the result to be a string of three `3`s, separated by dashes. Then we have one `1`. Then five `5`s.

More examples:

```
> :t nnn
nnn :: [Int] -> [[Char]]

> nnn [1,3..10]
["1","3-3-3","5-5-5-5-5","7-7-7-7-7-7-7","9-9-9-9-9-9-9-9-9"]

> nnn [10,2,4]
["10-10-10-10-10-10-10-10-10-10","2-2","4-4-4-4"]

> length (head (nnn [100]))
399
```

Note the math for the last example: 100 copies of `"100"` and 99 copies of `"-"` to separate them amount to 399 characters.

Assume that the values are greater than zero.

Remember: You can't write any recursive code!

Problem 8. (9 points) `expand.hs`

Consider the following two entries that specify the spelling of a word and spelling of forms of the word with suffixes:

```
program,s,#ed,#ing,'s
code,s,d,@ing
```

If a suffix begins with a pound sign (#), it indicates that the last letter of the word should be doubled when adding the suffix. If a suffix begins with an at-sign (`@`), it indicates that the last letter of the word should be dropped when adding the suffix. In all other cases, including the possessive (`'s`), the suffix is simply added. Given those rules, the two entries above represent the following words:

```
program
programs
programmed
programming
program's

code
codes
coded
coding
```

For this problem you are to write a function `expand entry` that returns a list that begins with the word with no suffix and is followed by all the suffixed forms in turn.

```
> :t expand
expand :: [Char] -> [String]

> expand "code,s,d,@ing"
["code","codes","coded","coding"]

> expand "program,s,#ed,#ing,'s"
["program","programs","programmed","programming","program's"]

> expand "adrift"              (If no suffixes, produce just the word.)
["adrift"]

> expand "a,b,c,d,e,f"
["a","ab","ac","ad","ae","af"]

> expand "a,b,c,d,@x,@y,@z,#1,#2,#3"
["a","ab","ac","ad","x","y","z","aa1","aa2","aa3"]

> expand "ab,#c,d,@e,f,::x"
["ab","abbc","abd","ae","abf","ab::x"]
```

A word may have any number of suffixes with an arbitrary combination of types. Words and suffixes may be arbitrarily long. You may assume that an entry never contains a blank, like `"a b,c"`.

Note that the only characters with special meaning are comma, `#`, and `@`. Everything else is just text.

Assume that the entry is well-formed. You won't see things like a zero-length word or suffix. Here are some examples of entries that will not be tested: `","`, `"test,"`, `"test,s,#,@"`

Remember: You can't write any recursive code!

Problem 9. (17 points) `pancakes.hs`

In this problem you are to print a representation of a series of stacks of pancakes. Let's start with an example:

```
> :t pancakes
pancakes :: [[Int]] -> IO ()

> pancakes [[3,1],[3,1,5]]
     ***
***   *
 *  *****
>
```

The list specifies two stacks of pancakes: the first stack has two pancakes, of widths `3` and `1`, respectively. The second stack has three pancakes. Pancakes are always centered on their stack. A single space separates each stack. Pancakes are always represented with asterisks.
Here's another example:

```
> pancakes [[1,5],[1,1,1],[11,3,15],[3,3,3,3],[1]]
                          ***
      *   ***********     ***
  *   *       ***         ***
***** * *************** *** *
>
```

There are opportunities for creative cooking:

```
> pancakes [[7,1,1,1,1,1],[5,7,7,7,7,5],[7,5,3,1,1,1],
[5,7,7,7,7,5], [7,1,1,1,1,1],[1,3,3,5,5,7]]
*******  *****  *******  *****  *******     *
   *     *******  *****  *******     *       ***
   *     *******   ***   *******     *       ***
   *     *******    *    *******     *      *****
   *     *******    *    *******     *      *****
   *      *****     *     *****      *     *******
>
```

Assume that there is at least one stack. Assume all stacks have at least one pancake. Assume all widths are odd numbers greater than zero. The smallest "order" you'll ever see is this:

```
> pancakes [[1]]
*
>
```

Like `street` on assignment 3, `pancakes` produces output. Use this structure:

```
pancakes stacks = putStr result
    where
         ...
```

```
        result = ...
```

Remember: You can't write any recursive code!

Problem 10. (17 points) `group.hs`

For this problem you are to write a program that reads a text file and prints the file with a line of dashes inserted whenever the first character on a line differs from the first character on the previous line. Additionally, the lines from the input file are to be numbered.

```
$ cat a4/group.1
able
academia
algae
carton
fairway
hex
hockshop

$ runghc group.hs a4/group.1
1 able
2 academia
3 algae
------
4 carton
------
5 fairway
------
6 hex
7 hockshop
$
```

First, note that the command `runghc`, not `ghci`, is being used.

Note also that only the lines from the input file are numbered. The separators are NOT numbered.

Lines with a length of zero (i.e., `length line == 0`) are discarded as a first step.

Another example:

```
$ cat a4/group.2
elemPos' _ [] = -1
elemPos' x ((val,pos):vps)
    | x == val = pos
    | otherwise = elemPos' x vps


f x y z = (x == chr y) == z

add_c x y = x + y


add_t(x,y) = x + y


fromToman 'I' = 1
```

```
fromRoman 'V' = 5
fromRoman 'X' = 10

p 1 (x:xs) = 10
$ runghc group.hs a4/group.2
1 elemPos' _ [] = -1
2 elemPos' x ((val,pos):vps)
------
3     | x == val = pos
4     | otherwise = elemPos' x vps
------
5 f x y z = (x == chr y) == z
------
6 add_c x y = x + y
7 add_t(x,y) = x + y
------
8 fromToman 'I' = 1
9 fromRoman 'V' = 5
10 fromRoman 'X' = 10
------
11 p 1 (x:xs) = 10
$
```

Note that when the line numbers grow to two digits the line contents are shifted a column to the right. That's ok.

If all lines start with the same character, no separators are printed.

```
$ cat a4/group.3
test
tests
testing
$ runghc group.hs a4/group.3
1 test
2 tests
3 testing
$
```

One final example:

```
$ cat a4/group.4
a
b
a
b
a
a
b
a
a
a
b
$ runghc group.hs a4/group.4
1 a
------
2 b
------
```

```
3 a
------
4 b
------
5 a
6 a
------
7 b
------
8 a
9 a
10 a
------
11 b
$
```

The separator lines are six dashes (minus signs).

Assume that there is at least one line in the input file. (A one-line file will produce no separators, of course.)

Implementation notes for `group.hs`

Unlike everything you've previously done with Haskell, this is a whole program, not just a function run at the `ghci` prompt. Follow the example of `longest` on slide 292 and have a binding for `main` that has a `do` block that sequences getting the command line arguments with `getArgs`, reading the whole file with `readFile`, and then calling `putStr` with result of a function named `group`, which does all the computation.

Your `group.hs` should look like this:

```
import System.Environment (getArgs)

main =
    do
       args <- getArgs
       bytes <- readFile $ head args
       putStr $ group bytes

...your functions here...
```

Yes, there's an `import` for something other than `Data.Char`. In this case we're asking for the `getArgs` function from the `System.Environment` module. This exception is permitted.

Note that the `$` operator, from slide 324, is being used to avoid some parentheses.

Remember: You can't write any recursive code!

Problem 11. (17 points) `avg.hs`

For this problem you are to write a Haskell program that computes some simple statistics for the hours reported in `observations.txt` submissions.

Let's use a pipeline to get a few lines of data into the file `avg.1`.

```
$ cat {...}/observations.txt | grep -i hours > a4/avg.1
```

Here's what we got:

```
$ cat a4/avg.1
Hours: 3-5
Hours: 10
I spent 8 hours on this.
Hours: 4-12
```

It looks like maybe somebody didn't read the instructions and wrote "`I spent...`" We'll ignore lines that don't start with "`Hours:`", case-sensitive. That leaves three lines, two with ranges. There's merit in being able to reflect uncertainty by reporting a range but we can't do simple arithmetic on a range. Let's view a range as representing three values: a low, a midpoint, and a high. Let's also view a single value as a range with low, midpoint, and high values that are equal. That gives us this view of the data:

	Low	Midpoint	High
3-5	3	4	5
10	10	10	10
4-12	4	8	12

Let's run `avg.hs` and specify only an input file:

```
$ runghc avg.hs a4/avg.1
n = 3
mean = 7.333
median = 8.000

Ignored:
Line 3: I spent 8 hours on this.
```

We see that:

- Three valid data points were found.
- The mean is `7.333`, which is `(4+10+8)/3` reported to three decimal places.
- The median is the middle data point in a sequence of values and in this case is `8`. As a simplification we show the median with three decimal places. (If there are an even number of values, and thus no middle value, the median is the mean of the two center-most values. For example the median of the four values `1`, `3`, `7`, `15` is `5` (`(3+7)/2`).)
- Line 3, whose contents are shown, was ignored.

If `avg.hs` is run with a `-l` (L) option, which must precede the file name, the low values (see the table) are used instead. In order, those values are `3`, `4`, `10`. We see this output:

```
$ runghc avg.hs -l avg.1
n = 3
mean = 5.667
median = 4.000
```

```
Ignored:
Line 3: I spent 8 hours on this.
```

Similarly, there's a `-h` option to compute the statistics using the high values (`5`, `10`, `12`):

```
$ runghc avg.hs -h avg.1
n = 3
mean = 9.000
median = 10.000

Ignored:
Line 3: I spent 8 hours on this.
```

Here are some points to keep in mind:

- Lines that don't start with "`Hours:`" are not included in the calculation but are reported under "`Ignored:`".

- Following "`Hours:`", discard all characters other than decimal digits, period (`.`), and dash (`-`).

  Then, ASSUME that what's left will be either a number, like `10`, or `7.5`, or a range, like `5.5-15`. (The behavior of `avg.hs` is undefined for other cases, which in practical terms means that I won't test with any such cases.) For ranges, the first value will always be less than the second.

- ASSUME that the command line arguments, which follow `runghc avg.hs`, are an optional `-l` or `-h`, followed by a file name. That amounts to three potential cases:

  ```
  runghc avg.hs FILENAME
  runghc avg.hs -l FILENAME
  runghc avg.hs -h FILENAME
  ```

  Behavior is undefined in all other cases. (Again, that means I won't test with any other cases.)

- ASSUME there will always be at least one valid data point in the input file.

- The tester's student set of tests for this problem will be the grading set of tests, too. In other words, **if the tester shows your `avg` passing all tests and you don't violate any restrictions, you are guaranteed full credit on this problem.** The final set of tests will be in place by noon on Saturday, Feburary 20.

Implementation notes for `avg.hs`

Here's a collection of implementation notes for `avg.hs`.

Some imports

In addition to the functions in the Prelude and `Data.Char`, you are permitted to use the following functions on this problem, `avg.hs`:

```
System.Environment.getArgs
Text.Printf.printf
```
and all functions in the `Data.List` module.

My solution starts by importing `getArgs` from `System.Environment`, `printf` from `Text.Printf`, and then all functions in `Data.List` and `Data.Char`:

```
import System.Environment (getArgs)
import Text.Printf (printf)
import Data.List
import Data.Char
```

See https://hackage.haskell.org/package/base-4.5.0.0/docs/Data-List.html for documentation on the functions in the `Data.List` module. Note: My solution uses only two functions from `Data.List`: `sort` and `partition`.

`main` for `avg.hs`

Just like `group.hs`, `avg.hs` does I/O. Here's the binding for `main` that I recommend you use:

```
main = do
    args <- getArgs
    bytes <- readFile $ last args
    putStr $ averages bytes $ init args
```

Like shown on slides 291-292, the `averages` function computes a string with newlines that `main` outputs with `putStr`. **Note that the `$` operator, from slide 324, is being used to avoid some parentheses.**

`Double` to fixed point conversion

Use the following function to convert `Doubles` to `Strings` with three places of precision, for mean and median.

```
fmtDouble::Double -> String
fmtDouble x = printf "%.3f" x
```

Dividing a `Double` by an `Int` (or `Integer`)

You'll find that dividing a `Double` by an `Int` or an `Integer` produces a type error. A conversion can be done with the `fromIntegral` function:

```
> let sum = read "10.4"::Double
> sum / (fromIntegral $ length [1,2])
5.2
```

The type of `fromIntegral` is interesting:

```
> :t fromIntegral
fromIntegral :: (Num b, Integral a) => a -> b
```

Rather than converting an `Integral` type to a specific type, like `Double`, it's treated as a more general thing, a type that's an instance of `Num`. Then in turn, that type can be converted to a `Double`.

A starter file

As a convenience, `a4/avg_starter.hs` has the above `imports` and `main`, a stub for `averages`, and `fmtDouble` from above.

splitHours

Also in in `a4/avg_starter.hs` is `splitHours`, a function to split up specifications of hours:

```
> splitHours "10"
["10"]

> splitHours "3.4-10.3"
["3.4","10.3"]
```

Another development/debugging technique

One way to get a look at values bound in a `where` clause for a function is to temporarily have the function return a tuple that comprises values of interest. Look at this **mid-development snapshot** of `averages`:

```
averages bytes args = (validEntries, "values:", values,
            "selected:", selected, "errors:", errs, stats, errors)
    where ...
          validEntries = ...
          values = ...
          selected = ...
          errs = ...
          stats = ...
          errors = ...
          ...and more...
```

Instead of returning a fully-formatted final result, it just creates a tuple with the various intermediate bindings like `validEntries`, `values`, `selected`, etc.

Let's try a call to it, passing in a string with embedded newlines, which might come from a two-line file, and the list `["-h"]`, simulating a `-h` command-line argument:

```
> averages "Hours: 10\nI spent 2 hours\n" ["-h"]
([(1,True,"10")],"values:",[(10.0,10.0,10.0)],"selected:",[10.0],"er
rors:",[(2,False,"I spent 2 hours")],"n = 1\nmean = 10.000\nmedian =
10.000\n","\nIgnored:\nLine 2: I spent 2 hours\n")
```

Note that the literal strings like `"values:"` just serve as labels, to help us see what's what. Note also that the `"stats:"` and `"errors:"` strings are the final output, in two pieces. You can learn a few things about how I approached the problem by looking closely at that output.

Below is a `main` that works with the mid-development snapshot of `averages` above. Because the version of `averages` above returns a tuple and `putStr` wants a string, I use `show` to turn that tuple into a string:

```
main = do
    args <- getArgs
    bytes <- readFile $ last args
    putStr $ (show $ averages bytes (init args)) ++ "\n"
```

With the above `averages` and `main`, here's what I see with my version:

```
$ runghc avg.hs -h avg.1
([(1,True,"3-5"),(2,True,"10"),(4,True,"4-12")],"values:",
```

```
[(3.0,4.0, 5.0), (10.0,10.0,10.0), (4.0,8.0,12.0)], "selected:",
[5.0,10.0,12.0], "errors:",[(3,False,"I spent 8 hours on this.")],"n
= 3\nmean = 9.000\nmedian = 10.000\n","\nIgnored:\nLine 3: I spent 8
hours on this.\n")
```

a4/tryall script

a4/tryall is a bash script that runs avg.hs on a given data file using each of the low, midpoint, and high modes in turn. Do cat a4/tryall to get a look at it and then do this:

```
$ a4/tryall a4/avg.1
...output for each of the three modes in turn...
```

Finally, one last reminder

Remember: You can't write any recursive code!

Problem 12. (ZERO points) rmRanges.hs

This problem is worth no points. Try it if you wish.

Write a function rmRanges that accepts a list of ranges represented as 2-tuples and produces a function that when applied to a list of values produces the values that do not fall into any of the ranges. Ranges are inclusive, as the examples below demonstrate.

Note that rmRanges is typed in terms of the Ord type class, so rmRanges works with many different types of values.

```
> :type rmRanges
rmRanges :: Ord a => [(a, a)] -> [a] -> [a]

> rmRanges [(3,7)] [1..10]
[1,2,8,9,10]

> rmRanges [(10,18), (2,5), (20,20)] [1..25]
[1,6,7,8,9,19,21,22,23,24,25]

> rmRanges [] [1..3]
[1,2,3]

> rmRanges [('0','9')] "Sat Feb  8 20:34:50 2014"
"Sat Feb   :: "

> rmRanges [('A','Z'), (' ', ' ')] it
"ateb::"

> let f = rmRanges [(5,20),(-100,0)]

> f [1..30]
[1,2,3,4,21,22,23,24,25,26,27,28,29,30]

> f [-10,-9..21]
[1,2,3,4,21]
```

Assume for a range (x, y) that x <= y, i.e, you won't see a range like (10,1) or ('z', 'a').

As you can see above, ranges are inclusive. The range `(1,3)` removes `1`, `2`, and `3`.

Just for fun...Here's an instance declaration from the Prelude:

```
instance (Ord a, Ord b) => Ord (a, b)
```

It says that if the values in a 2-tuple are orderable, then the 2-tuple is orderable. With that in mind, consider this `rmRanges` example:

```
> rmRanges [((3,'z'),(25,'a'))] (zip [1..] ['a'..'z'])
[(1,'a'),(2,'b'),(3,'c'),(25,'y'),(26,'z')]
```

Remember: You can't write any recursive code!

Problem 13. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one `"Hours:"` line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a4/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `aN.YYYYMMDD.HHMMSS.tz` You can run `a4/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `372s16` if you need to recover a file and aren't familiar with `tar`—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a4/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
$ wc $(grep -v txt a4/delivs)
  26  129  472 warmup.hs
   4   22  103 dezip.hs
   2   13   54 repl.hs
   2   12   58 doubler.hs
   2    9   68 revwords.hs
   7   30  131 cpfx.hs
   5   36  212 nnn.hs
  19   84  536 expand.hs
  16   81  551 pancakes.hs
  19   84  554 group.hs
  58  317 2022 avg.hs
   9   54  306 rmRange.hs
 169  871 5067 total
```

My code has few comments.

Miscellaneous

REMEMBER: Except for `warmup.hs` you are not permitted to write any recursive functions on this assignment!

This assignment is based on the material on Haskell slides 1-354.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem would correspond to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 10 to 15 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put ten hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached ten hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.