# CSC 372, Spring 2015
# Assignment 3
# Due: Thursday, February 26 at 23:59:59

## ASSIGNMENT-WIDE RESTRICTIONS

There are three assignment-wide restrictions:

1. Minus a couple of 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 useful and interesting but we haven't covered them. I want your attention focused on the elements of Haskell that we have covered.

3. Recall the idea put forth on slide 224: 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 `foldl`.

**Use the tester!**

Just like for assignment 2, there's a version of `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! I'll be happy to help you with using the tester and understanding its output.

I recommend you make these two symlinks in your assignment 3 directory on lectura:

```
$ ln -s /cs/www/classes/cs372/spring15/a3 a3
$ ln -s a3/tester t
```

The first link, `a3`, is referenced by second link, `t`, and is also handy for accessing data files in my `a3` directory.

On assignment **2** I had you running the tester like this:

```
$ a2/tester warmup
```

With the `t` symlink to `a3/tester` you'll be able to run the assignment 3 tester like this:

```
$ t warmup
```

If you see get `t: command not found` that's probably because `.` (dot) isn't in your `$PATH`, the list of directories that the shell searches for commands. The short answer is to just use `./t warmup` instead. (The medium length answer is that having dot at the end of your path is handy and reasonably safe. See "To Dot or Not To Dot" on the resources page for a long story about having dot in your path, and where.)

**Problem 1. (6 points) `warmup.hs`**

This problem is like `warmup.hs` on assignment 2—I'd like you to write your own version of some functions from the Prelude: `map`, `filter`, `foldl`, `foldr`, `any`, and `all`.

The code for `map`, `filter`, and `foldr` is in the slides and the others are easy to find, but I'd like you 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` |
| `fldl` | `foldl` |
| `fldr` | `foldr` |
| `myany` | `any` |
| `myall` | `all` |

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!

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

**Problem 2. (2 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`:**

Your solution may only use `map`, `fst`, the composition operator, and this function:

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

Remember that writing recursive function 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 253-254), create a function `revwords` that reverses the words in a sentence. Assume the sentence contains only letters and spaces.

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

> revwords "testing"
"gnitset"

> revwords ""
""
```

**Problem 6. (2 points) `cpfx.hs`**

`a3/whm_cpfx.hs` is my solution for `cpfx` from assignment 2. 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 2 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.

Assume that the values are greater than zero.

**Remember: You can't write any recursive code!**

**Problem 8. (10 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 at-sign (`@`), it indicates that the last letter of the word should be dropped when adding the suffix. In other cases, including the possessive (`'s`), the suffix is simply added. The above two entries 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 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.

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. (16 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 greater than zero. The smallest order you'll see is this:

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

To avoid problems with centering, assume all pancake widths are odd.

Like `street` on assignment 2, `pancakes` produces output. Use this structure:

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

**Remember: You can't write any recursive code!**

**Problem 10. (15 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 group.1
able
academia
algae
carton
fairway
hex
hockshop

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

Note 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 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 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 group.3
test
tests
testing
$ runghc group.hs group.3
1 test
2 tests
3 testing
$
```

One final example:

```
$ cat group.4
a
b
a
b
a
a
b
a
a
a
b
$ runghc group.hs 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.)

`a3` has the `group.[1234]` files shown above. If you're on lectura and have made the `a3` symlink, you can create symlinks to those files like this:

```
$ ln -s a3/group.* .
```

Once that's done, you'll can reference the files like this:

```
$ runghc group.hs group.4
```

You could also reference them using the `a3` symlink:

```
$ runghc group.hs a3/group.4
```

**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 247 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 to run `group.hs`, you use `runghc` on the command line. If you've installed The Haskell Platform for Windows using the defaults, I believe you'll find that you have a `runghc` command at your disposal. Here's what running it in a Windows "cmd" window looks like:

```
W:\372>runghc group.hs group.1
1 able
2 academia
3 algae
------
4 carton
------
5 fairway
------
6 hex
7 hockshop

W:\372>
```

Note that the blank line following "`7 hockshop`" is produced by Windows, not `group`.

**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 rough cut of the data into the file `avg.1`.

```
$ cat */observations.txt | grep -i hours > avg.1
```

Here's what we got:

```
$ cat 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 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 (`-`).
- 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.

- For this problem the full set of tests used for grading will be exercised by `a3/tester`. **If `t avg` shows all tests as passing 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 21.

**Implementation notes for `avg.hs`**

Use this function to convert `Double`s to `String`s with three places of precision, for mean and median.

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

Here's how my solution starts:

```
import System.Environment (getArgs)
import Data.List (sort, partition)
import Text.Printf (printf)
```

On this problem, `avg.hs`, in addition to using the non-Prelude functions cited above (`getArgs`, `sort`, `partition`, and `printf`), you can use any of the functions in `Data.List`. (See http://hackage.haskell.org/package/base-4.7.0.2/docs/Data-List.html)

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

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

Like the `longest` function shown on slides 246 and 247, the `averages` function computes a string with newlines that `putStr` simply outputs. Note that the `$` operator, from slide 264 is being used to avoid some parentheses.

In Piazza post @75 I write, "One way to get a look at bindings developed in a where clause for a function is to temporarily have the function return a tuple of value of interest.", and then show some examples. Here's the form that might take on this problem:

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

```
            stats = ...
            errors = ...
            ...and more...

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

Note that this `main` is a little different from the first `main` shown above: because the development / debugging version of `averages` above returns a tuple and `putStr` wants a string, I use `show` to turn that tuple into a string.

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")
```

Note that the literal strings like `"values:"` just serve as labels, to help me 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.

As a convenience, `a3/avg_starter.hs` has imports, a `main`, a stub for `averages`, and `fmtDouble` from above.

My solution orginally used `Data.List.Split.splitOn` to break up strings like `"10-20"` but minutes before finalizing this write-up I found that package is not installed on lectura. `a3/avg_starter.hs` has this simple splitter that should meet your needs:

```
> splitValue "10"
["10"]

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

`a3/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 a3/tryall` to get a look at it and then do this:

```
$ a3/tryall a3/avg.1
...output for each of the three modes in turn...
```

Just like for `group.hs`, you might create symlinks to the data files like this:

```
$ ln -s a3/avg.* .
```

**Remember: You can't write any recursive code!**

**Problem 12. (2 points) `guards.txt`**

Slide 153 says, "*If values in a call match the pattern(s) for a clause and a guard is true, the corresponding expression is evaluated*." When presenting that slide I wondered if anybody would ask this question:

> "What happens if a pattern matches but no guard is true?"

**For this problem you are to create a plain ASCII draft of a single "slide" that answers that question**. Follow the style of my slides and language-related Piazza posts: Present a mix of sentences, Haskell code, and interaction with that code. As a ballpark figure, think in terms of about 15-25 lines, some blank.

This problem is an exercise in learning how to learn a language. When learning a language you'll often come to questions that aren't easily answered by reading the documentation or searching the web. In those cases it's often possible to devise an experiment to resolve the question. As a trivial example, consider the question "What does `take` do with a negative `n`?" It's easy to try `take -5 [1..3]` to explore that behavior.

Approach this problem by creating a simple example where a function value matches the pattern for a clause but fails the test(s) for the guards of that clause. What happens then? If your answer is as simple as "it's an error", you've probably overlooked something.

The goal is to learn by experimentation so if your first step is to Google or read the docs, you've missed the point!

Important: `guards.txt` must be plain ASCII text; DO NOT turn in a PowerPoint file!!

**Problem 13. (ZERO points) `mfilter.hs`**

**This problem is worth no points**, like `whynot.txt` on assignment 1, but this one is significantly harder. 58 out 70 students submitted a solution for `whynot.txt`. Let's see who will try this one.

Write a function `mfilter` 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 `mfilter` is typed in terms of the `Ord` type class, so `mfilter` works with many different types of values.

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

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

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

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

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

> mfilter [('A','Z'), (' ', ' ')] it
```

```
"ateb::"

> let f = mfilter [(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. A range like `(1,3)` blocks `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 `mfilter` example:

```
> mfilter [((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 14. 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 for the whole assignment on a line by itself, more or less like one of these:

```
Hours: 10
Hours: 11.2-15.8
Hours: 15+
```

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, "Interesting!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

### Turning in your work

Use the D2L Dropbox named `a3` to **submit a single zip file named `a3.zip` that contains all your work**. If you submit more than one `a3.zip`, your final submission will be graded. Here's the full list of deliverables:

```
warmup.hs
dezip.hs
repl.hs
doubler.hs
```

`revwords.hs`
`cpfx.hs`
`nnn.hs`
`expand.hs`
`pancakes.hs`
`group.hs`
`avg.hs`
`guards.txt`
`mfilter.hs` (zero points!)
`observations.txt` (for extra credit)

Note that all characters in the file names are lowercase.

`

**Miscellaneous**

This assignment is based on the material on Haskell slides 1-285.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds 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.) Two minus signs (`--`) is comment to end of line; `{-` and `-}` are used to enclose block comments, like `/*` and `*/` in Java.

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 12 hours to complete this assignment, **assuming** he or she completed all problems on assignment 2.

**Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help.** Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems. If you reach the eight-hour mark, regardless of whether you have specific questions, it's probably time to touch base with us. Give us a chance to speed you up! **Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.**

When developing code that performs a complex series of transformations on data I recommend the technique shown for `longest`, starting on slide 226. That is, work through a series of `lets` to check the transformation made by each step.

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, and more. (See the syllabus for the details.)

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