CSC 372, Spring 2014
Assignment 2
Due: Wednesday, February 26 at 23:00

ASSIGNMENT-WIDE RESTRICTIONS

There are two assignment-wide restrictions:

1. The only module you may use is `Data.Char`.

2. Except for problem 1, `warmup.hs`, **you may not write any recursive functions!** Instead, use higher-order functions like `map`, `filter`, and `foldl`.

   Note that there is both direct and indirect recursion. With direct recursion, a function `f` calls itself. With indirect recursion, `f` might call `g` and then `g` might call `f`.

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

This problem is like `warmup.hs` on assignment 1—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` are in the slides; 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. (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 problem restrictions 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 []
([],[])
```

RESTRICTIONS for `dezip.hs`:

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

```
swap (x,y) = (y,x)
```

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"]
```

RESTRICTIONS 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. (3 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 create `doubler` using 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 (slide 240), 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. (3 points) `cpfx.hs`

`http://www.cs.arizona.edu/classes/cs372/spring14/a2/whm_cpfx.hs` is my solution for `cpfx` from assignment 1. The `cpfx` function is recursive. Rewrite that function to be non-recursive but still make use of the `cpfx'` function.

See the assignment 1 write-up for examples of using `cpfx`.

Problem 7. (10 points) `mfilter.hs`

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: No explicit recursion!

NOTE: This problem requires a somewhat creative leap. I won't say more about that here so I don't spoil it for anybody but don't feel bad about asking for a hint on this one.

Problem 8. (10 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: No explicit recursion!

Problem 9. (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 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 we will **not** test with: `","`, `"test,"`, `"test,s,#,@"`

Remember: No explicit recursion!

Problem 10. (18 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 1, `pancakes` produces output. (Now we know to say that `street` and `pancakes` are functions that produce *actions*–see slide 276 and following.) My solution is structured like this:

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

Remember: No explicit recursion!

Problem 11. (18 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 numbed—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.)

`http://www.cs.arizona.edu/classes/cs372/spring14/a2` has the `group.[1234]`

files shown above. (That's `/cs/www/classes/cs372/spring14/a2` if you're on lectura.)

Implementation note: Unlike everything else you've done with Haskell, this is a whole program, not just a function run at the `ghci` prompt. Follow the example of `longest` on slides 233-234 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` might look like this:

```
import System.Environment (getArgs)

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

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

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 at a Windows XP command prompt 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`.

If you're working on your own machine I recommend that you do a simple test of `runghc` ASAP, so that if problems turn up, we'll have plenty of time to help you get it working. Just try a version check:

```
W:\372>runghc --version
runghc 7.6.3
```

Remember: No explicit recursion!

Problem 12. 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 these:

```
Hours: 10
Hours: 12-15.5
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 `a2` to **submit a single zip file named `a2.zip` that contains all your work**. If you submit more than one `a2.zip`, we'll grade your final submission. Here's the full list of deliverables:

`warmup.hs`
`dezip.hs`
`repl.hs`
`doubler.hs`
`revwords.hs`
`cpfx.hs`
`mfilter.hs`
`nnn.hs`
`expand.hs`
`pancakes.hs`
`group.hs`
`observations.txt` (for extra credit)

DO NOT SUBMIT INDIVIDUAL FILES—submit a file named `a2.zip` that contains each of the above files.

Note that all characters in the file names are lowercase.

Have all the deliverables in the uppermost level of the zip. It's ok if your zip includes other files, too.

Miscellaneous

This assignment is based on the material on Haskell slides 1-280.

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 15 hours to complete this assignment.

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 `let`s 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!