CS212: Paradigms
Dr. Gross, Spring 2016

Lab 11 - See Lab 11 for Information

1. Create a new package called haskell.lastnamelastname
2. Copy the files from the Lab 11 directory (Documents\Lab11) to the package above
3. When you start on your homework, copy your files your homework repo into a new package: haskell.lastname

Some useful Haskell references:
http://www.one-tab.com/page/3qZ4eyDoRq6NR1JagK_-zw
Warning: you may (will) get these errors:

```
<interactive>:6:12: lexical error at character '\SYN'
<interactive>:21:9: lexical error at character '\ESC'
```

ghci is very sensitive to character input. Try again and **don't copy & paste**.

Due Thursday, April 28 before lab

Part 1: Fix the Bug (Everyone Finished This, Don't Need to Redo)

The code in factor.hs Documents.Lab11 has a bug. It crashes. Yes, sure, by all means fix the bug, but before you are done you need to be able *to explain what caused the bug in the first place.*

Some of you noticed that I spelled out the nature of the problem in the commit message for the code (which you would have seen when you pulled the lab repo from upstream). This is not me being a jerk (not that I'm saying I'm not a jerk; it's a reminder that commit messages contain important information, and that to be a successful computer scientist you need to pay attention to detail. The problem was essentially in the case where y == 2. Groups that saw this message spent less time diagnosing.

Part 2: Write Some Haskell

Create a file called lab11.hs in haskell.lastname in your homework repo and write the following functions. *Here's a hint: not every one of these should necessarily be written in the order listed here. Read through them and if you see a function that can call another function, write the second function first.* ***Don't repeat code any more than necessary (see rubric).*** **To avoid namespace conflicts, add fl to the end of your functions, where f is your first initial, and n is your last initial** (so foo becomes foojb for me) or call them foo' (foo prime).

2A Simple integer to boolean functions

Some functions in Haskell take in an integer value and return a boolean. Let's write some!

1. Returns True if x is even, or False if x is odd
   `isEven x`
2. Returns False if the number is even, or True if the number is odd
   `isOdd x`

Once you have one of these functions, the other is cakewalk *and won't even use a condition or True or False*. In fact, there are two ways to write this. See if you can figure out both.

2B Simple integer to integer functions

Some functions in Haskell take in a value (we're only working with integers) and returns a new (modified) value. This does NOT mean there's no recursion. For example, factorial x (aka x factorial or x!) multiplies x times each number between 1 and x-1.
Simple functions:

- Adds one to x and returns that number
  `increment x`

- Subtracts one from x and returns that number
  `decrement x`

2C Simple list to integer functions

Here are some important list functions:

- Adds each element of x and returns that sum
  `sum x`

- Multiplies each element of x and returns that product
  `product x`

- Returns the number of elements in x
  `count x`

2D Simple integer to list functions

- Returns next x even numbers
  `evensTo x`

- Returns next x odd numbers
  `oddsTo x`

2E Simple list to list functions

- returns a list of the elements in x that are even
  `evens x`

- Returns a list of elements in x that are even
  `odds x`

2A' Harder integer to boolean functions

- Returns True if x is prime or False if x is not prime
  `isPrime x`

- Returns True if the given number is in the Fibonacci sequence or False if x is not in that sequence
  `isFibonacci x`

2B' Harder integer to integer functions

- Returns x multiplied each integer between x-1 and y
  `partialFactorial x y`

- Returns the next prime number >= x
  `nextPrime x`

- Returns the next number in the Fibonacci sequence >= x
  `nextFibonacci x`

2C' Harder list to integer functions

- Adds each even element to the total and multiplies the total by each odd element
  `sumEvensProductOdds x`

- Divides the sum of each element of x by the count of each element of x
  `mean x`

2D' Harder integer to list functions

- returns a list of the first x prime numbers
  `primes x`

- Returns a list of the first x numbers in the fibonacci sequence
  `fibonaccis x`

- Returns a list of prime numbers <= x
  `primesTo x`

- Returns a list of numbers in the Fibonacci sequence <= x
  `fibonaccisTo x`

2E' Harder list to list functions

- Returns a list of all elements of x that are primes
  `selectPrimes x`

- Returns a list of all elements of x that are in the Fibonacci sequence
  `selectFibonaccis x`

3: Function Passing

These two are simple. Or rather, deceptively simple. They look easy, and like lots of knowledge, they're a total cakewalk to write *once you know how to write them*.

Write a function that constructs and returns a list by applying function f to each element of the input list x for which the function test returns true:

```
applyToSome test f x
```

It's useful to have a subfunction that applies function f to x and returns that value:

```
apply f x
```

Then you'll just need to make applyToSome's complement, which is that constructs and returns a list by applying function f to each element of x that function test does not return true for. This is a much simpler function to write *because you'll use applyToSome*.

```
applyToOthers f x
```

In order to test this code, you'll need both a function that takes in a number and returns a boolean, and another function that takes in a number and returns a different number. Hmm, if only you had already some of this written.

4: List Comprehensions

Remember that to define a list, all I need is a label, an assignment, and the list. I don't have to tell Haskell the list (although I can), I can tell Haskell the rules of the list and have Haskell do the work for me. In ghci, I use the let construct:

```
let x = [1..20]
```

I can use let x = again, but that causes a reload and I'll likely lose everything else. I can create a list in my .hs file the same way I create a function *because functions are first order objects*. They're at the top of the heap. When all you have is a functional language, everything looks like a function. The first type of list comprehension (a function that defines a list) just looks like convenient shorthand:

```
x = [1,3..99]
```

This will set x to a list containing all of the odd numbers between 1 and 99. What would happen if I didn't have the ",3"? Try out both.

Your first list comprehension should be the number 99 down to three (in that order!). Call this list threes.

```
threes = …
```

The next kind of list comprehension takes the form of a set comprehension. The tutorial *Learn You a Haskell* says "if you've ever taken a course in mathematics, you've probably run into set comprehensions." I doubt this is true. A set is not exactly the same as a list, but don't worry about that for now. You'll learn about sets and set comprehensions in Discrete Math (and again in Data Structures), but here's an intro:

$$S = \{\textit{some operation on } x | \, x \in [], \textit{some boolean on } x\}$$

This may (or may not) look evil. It's not, it's just… intense. It means the set S contains all numbers that

- Are members of N (N is all of the natural numbers, and ∈ means "contained in"),
- Pass the boolean test at the end
- Are altered by the operation at the beginning

So if I wanted the squares of all of the multiples of three, I would write (in set notation):

$$S = \{2x \mid x \in N, x \,\%\, 3 == 0\}$$

or in Haskell (adding in the constraint that I only want to start with the numbers between 1 and 1000):

```
let s = [x * 2 | x <- [1..1000], x `mod` 3 == 0]
```

Let's break this down, because this is all stuff you've seen before. The x here is a local variable inside the list comprehension (which, remember, *is a function,* so can have local variables). The first part that happens is the middle, [1..1000]. That's the generation of the numbers 1 through 1000 (inclusive). Then each number is compared to the Boolean operation after the comma, which obviously returns true only for multiples of three. Finally, we're going to double each of those numbers and end up with s:
[6, 12, 18...

A note: Haskell can and will create endless lists. It can do so because it's lazy. In this case, lazy is a technical term. Haskell being lazy means that it doesn't actually do the work you want done until the last possible moment. Ha-ha, just like students, I'm so very funny. Anyway, laziness can mean that I can use a function like `take x y` (which returns the first x elements of the list y) to grab the contents of an infinite list. Alan Turing would be weeping. The point is that it's possible to inadvertently create an infinite list, just as you can recurse or loop infinitely. Fair warning. Won't necessarily happen.

Here are some more fun lists to make (write each one as a separate expression in your file):

1. The the odd numbers from 1 to 100
2. The odd numbers from 1 to 100 that are divisible by three
3. The odd numbers from 1 to 100 that are divisible by three or five
4. The square of the odd numbers from 1 to 1,000 that are divisible by three or five but not by seven

Rubric

Standards/comments (yes, *true* functional programmer don't comment, but you will for now)	5
Time estimate/accounts	5
All functions written	30
All functions correct	10
No repeated code (or as little as possible; this may mean writing other functions not listed)	10
Functions for function passing written	10
Function passing demonstrated	10
All list comprehensions written	10
All list comprehensions correct	10
Total	**100**
Not asking Dr. Gross a question in office hours or email	-100