Homework 3 Due March 20th 11:59 pm

Submission rules:

- All text answers must be given in Haskell comment underneath the problem header.
- You must submit a single .hs file with the following name: firstName-lastName-hw3.hs. Failure to do so will result in -10 points.
- You will lose 10 points if you put a module statement at the top of the file.
- You will lose 10 points for any import statements you have in your file and will automatically miss any problems you used an imported function on.
- If your file doesn't compile you will lose 10 points and miss any problems that were causing the compilation errors.
- This means that any function which is causing compiler errors should be commented out. There will be no partial credit.
- You must use the skeleton file provided and must not alter any type signature. If you alter a type signature you will automatically miss that problem.
- You will lose 10 points if you include a *main* function in your file.

Problems

Problem 1 (Exercise 7.5) (2 pts)

Without looking at the definitions from the standard prelude, define the higher-order library function *curry'* that converts a function on pairs into a curried function, and, conversely, the function *uncurry'* that converts a curried function with two arguments into a function on pairs.

Hint: first write down the type of the two functions.

Problem 2 (Exercise 7.9) (5 pts)

A higher-order function *unfold* that encapsulates a simple pattern of recursion for producing a list can be defined as follows:

```
unfold :: (b -> Bool) -> (b -> a) -> (b -> b) -> b -> [a]
unfold p h t x | p x = []
               | otherwise = h x : unfold p h t (t x)
```

That is, the function *unfold p h t* produces the empty list if the predicate *p* is true of the argument value, and otherwise produces a non-empty list by applying the function *h* to this value to give the head, and the function `t` to generate another argument that is recursively processed in the same way to produce the tail of the list. For example, the function *int2bin* can be rewritten more compactly using *unfold* as follows:

```
int2bin = unfold (== 0) (`mod` 2) (`div` 2)
```

Redefine the functions *chop8*, *map' f* and *iterate' f* using *unfold*.

```
> chop8 [1..10]
[[1,2,3,4,5,6,7,8],[9,10]]
> chop8 [1..20]
[[1,2,3,4,5,6,7,8],[9,10,11,12,13,14,15,16],[17,18,19,20]]

chop8 :: [a] -> [[a]]
chop8 = unfold ...

> map' (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]

map' :: (a -> b) -> [a] -> [b]
map' f = unfold ...

> take 10 (iterate' (+1) 0)
[0,1,2,3,4,5,6,7,8,9]
```

```
iterate' :: (a -> a) -> a -> [a]
iterate' f = unfold ...
```

Problem 3 (4 pts)

Define a function, *concat*, which takes a list of lists and transforms it into a flattened list. Write this function 3 different ways:

1. Using explicit recursion
2. Using *foldr*
3. Using *foldl*

Name the functions as follows:

1. *concatER*
2. *concatFR*
3. *concatFL*

All functions should have the following type signature:

```
concat## :: [[a]] -> [a]
```

Examples:

```
> concatER [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,4,5,6,7,8,9]
> concatFR [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,4,5,6,7,8,9]
> concatFL [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,4,5,6,7,8,9]
```

Problem 4 (5 pts)

Write a function, *disjunction2*, which takes two predicates as arguments and returns the predicate which returns *True* if either of the two predicates returns *True*. For example:

```
> disjunction2 even odd 3
True
> disjunction2 even odd 4
True
> disjunction2 (> 5) (> 10) 3
False

disjunction2 :: (a -> Bool) -> (a -> Bool) -> a -> Bool
```

Problem 5 (5 pts)

Use *foldr* to write a function, *disjunction*, which takes an arbitrary number (> 0) of predicates as arguments. For example:

```
> disjunction [even, odd, (> 5)] 3
True
> disjunction [even, odd, (> 5)] 4
True
> disjunction [(> 10), (> 5)] 4
False

disjunction :: [a -> Bool] -> a -> Bool
disjunction ps x = foldr ...
```

Problem 6 (7 pts)

Use *foldr* to write a function, *deleteDupes*, which takes a list and returns a list with all duplicate elements removed. Note you must use *foldr* but you are also free to use other helper/higher-order functions. For example:

```
> deleteDupes [1,2,2,3,1,4]
[1,2,3,4]
> deleteDupes [4,3,3,4,1,2,2]
[4,3,1,2]
> deleteDupes [1,2,3,4,4,5,6,6,6,7,8,8,9]
[1,2,3,4,5,6,7,8,9]
> deleteDupes "abracadabra"
"abrcd"

deleteDupes :: Eq a => [a] -> [a]
deleteDupes xs = foldr ...
```

Problem 7 (7 pts)

Using *foldl* write a function, *tally*, which returns the number of elements that pass a predicate. For example:

```
> tally even [1..10]
5
> tally odd [1..10]
5
> tally (> 5) [1..10]
5

tally :: (a -> Bool) -> [a] -> Int
tally p xs = foldl ...
```

Problem 8 (7 pts)

Using *foldr* and *zip* write a function, *bangBang*, which takes a list and returns the *nth* element of the list. You can assume the list is non-empty and if *n* is larger than the list then return the last element of the list. For example:

```
> bangBang [1..10] 3
4
> bangBang [1..10] 5
6
> bangBang [1,2,3] 0
1
> bangBang [1..5] 10
5

bangBang :: [a] -> Int -> a
bangBang xs n = foldr ... zip ...
```

Problem 9 (8 pts)

Using *foldr* and *zip* write a function, *increasing*, which takes a list and determines if the list is in increasing order. For example:

```
> increasing [1,2,3,4,5]
True
> increasing [1,1]
True
> increasing [1,2,1]
False

increasing :: Ord a => [a] -> Bool
increasing xs = foldr ... zip ...
```

Problem 10 (10 pts)

Using *foldl* write a function, *decimate*, which takes a list and removes every *10th* element. For example:

```
> decimate [1..21]
[1,2,3,4,5,6,7,8,9,11,12,13,14,15,16,17,18,19,21]

decimate :: [a] -> [a]
decimate xs = foldl ...
```

Problem 11 (10 pts)

Define a function, *encipher*, which takes two lists of equal length and a third list to encrypt. It uses the first two lists to define a substitution cipher which it uses to encrypt the third list. For example:

```
> encipher ['A'..'Z'] ['a'..'z'] "HELLO"
"hello"
```

```
encipher :: Eq a => [a] -> [b] -> [a] -> [b]
```

Problem 12 (10 pts)

Define a function, *prefixSum*, which takes a list and returns a list of all sums of prefixes of that list. For example:

```
> prefixSum [1,2,3,4]
[1,3,6,10]
> prefixSum [1..10]
[1,3,6,10,15,21,28,36,45,55]
> prefixSum [2,5]
[2,7]
```

```
prefixSum :: Num a => [a] -> [a]
```

Problem 13 (20 pts)

Define a function, *minesweeper*, which takes a non-empty list of *String* and returns a non-empty list of *String* with the number of mines adjacent to each cell. Each mine is represented with the * character and empty space the . character. If the space has no mines near it then leave it empty. You may need the function intToDigit:

```
intToDigit :: Int -> Char
intToDigit 0 = '0'
intToDigit 1 = '1'
intToDigit 2 = '2'
intToDigit 3 = '3'
intToDigit 4 = '4'
intToDigit 5 = '5'
intToDigit 6 = '6'
intToDigit 7 = '7'
intToDigit 8 = '8'
intToDigit 9 = '9'
```

For example:

```
> minesweeper ["*..",
               "..*",
               "..."]
["*21",
 "12*",
 ".11"]
```

```
minesweeper :: [String] -> [String]
```