Homework 2  
COSC 3015  
Due: 9/23/2014

**Part 1**

In class we introduced an implementation of quicksort.

\[
\text{qsort} \[ \] = \[
\text{qsort} \ (x:xs) = \text{qsort} \ \text{smaller} ++ [x] ++ \text{qsort} \ \text{larger}
\]

\[
\text{where smaller} = [ \ a \ | \ a < xs, a <= x] \\
\text{larger} = [ \ b \ | \ b < xs, b > x]
\]

The type of this function, inferred in ghci, is

\[
\text{qsort} :: (\text{Ord} \ a) \Rightarrow \[a\] \rightarrow [a]
\]

which means qsort is a function from the type of lists containing type \(a\) to the type of lists containing type \(a\), where type is constrained by some ordering relation, which is indicated by the type class modifier \(\text{Ord} \ a\) (in other words for any \(x\) and \(y\) in type \(a\), \(x \leq y\) can be determined).

In class we gave examples of sorting integers however the above implementation will work on any type that has an ordering relation: Char, Float, String ([Char]), Integers and so on.

What if we wanted to constrain an implementation only to numeric types and make sure that our implementation was constrained on type \(a\) to be in the type class \(\text{Num} \ a\)? One quick way to do this would be to implement some arithmetic function within the definition.

\[
\text{qsort'} \ [ ] = [ ] \\
\text{qsort'} \ (x:xs) = \text{qsort'} \ \text{smaller} ++ [x] ++ \text{qsort'} \ \text{larger}
\]

\[
\text{where smaller} = [ \ a+1 \ | \ a < xs, a <= x] \\
\text{larger} = [ \ b+1 \ | \ b < xs, b > x]
\]

Note that we have modified the list comprehension to add 1 to each element of the remaining list. Hence we have now assured that the type over which qsort’ operates must be a numeric value and the resulting type is:

\[
\text{qsort'} :: (\text{Ord} \ a, \text{Num} \ a) \Rightarrow \[a\] \rightarrow [a]
\]

However there is a problem with this implementation….

*Main> qsort’ (reverse [1..10])  
[10,10,10,10,10,10,10,10,10,10]

*Main> qsort’ [4,6,7,2,1,8,3,6]  
[3,3,5,4,8,7,9,11]

*Main> qsort’ [1,10,9,2,8,3,7,4,6,5]
Something odd is happening in the evaluation of the list comprehensions. What is it? How can we fix it and maintain the assertion that the type \( a \) must be in the type class \( \text{Num} \)?

Here is repaired version, called \( \text{qsort1} \), showing the tests cases run above.

Assignment:
1) Explain the behavior of \( \text{qsort}' \). (Describe how the evaluation is causing the bad results. It will help to walk through a couple of steps by hand.)
2) Write a version of \( \text{qsort} \) (\( \text{qsort1} \)) that has the correct type and sorts the lists appropriately.
3) Run the test cases to verify these results.
Hand in, via email to the instructor, all of your findings.

**Part 2**
(The topics in this part will be presented in class on Tuesday (9/16), although much of it will be familiar from Thursdays discussion.)

In class we discussed the following code.

\[
\begin{align*}
\text{plus} & : (\text{Integer}, \text{Integer}) \rightarrow \text{Integer} \\
\text{plus} \ (x, y) &= x + y \\
\text{plusc} & : \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{plusc} \ x \ y &= x + y
\end{align*}
\]

The function \( \text{plus} \) takes its arguments all at once packaged in a pair while \( \text{plusc} \) takes its arguments one at a time. We discussed how Haskell supports a notation for describing a function without forcing you to choose a name for it.
The general form is
\[
\backslash x \rightarrow e
\]
where \( x \) is a variable and \( e \) is a Haskell expression.
Note that in Haskell “→” is used to denote the type constructor for functions (e.g. if γ and δ are types, then γ → δ is the type of functions from γ to δ. Also, “→” is used in the expression language to describe an actual function, (x → e) denotes a function whose single argument is referred to in the expression e by the variable x.

This overloading of syntax is similar to that for Cartesian products. If γ and δ are types then (γ,δ) is the type whose elements are the pairs where the first element comes from γ and the second element comes from δ. But also, if a ∈ γ and b ∈ δ, then the pair (a,b) ∈ (γ,δ). So the developers of Haskell have used the same notation for the type constructor and to construct the elements of the type in both cases.

Now, consider the following interaction with the Haskell interpreter.

```
Main> :t plusc
    plusc :: Integer -> Integer -> Integer
Main> :type plusc 7
    plusc 7 :: Integer -> Integer
```

Evidently plusc 7 is a function of type Integer -> Integer. But what function is it? It is the function that is expecting an input y and will compute the expression 7 + y. So, it is the function described by the following expression:

\( \lambda y \rightarrow 7 + y \)

This form of Haskell expression is called a *lambda-term* (\(\lambda\)-term).

We can write \(\lambda x y \rightarrow e\) for \(\lambda x \rightarrow (\lambda y \rightarrow e)\).

Every function can be written in a form where no arguments are declared on the left side of the definition. If e is an arbitrary Haskell expression, then the following examples show how this works.

- \(f x = e\) is the same as \(f = \lambda x \rightarrow e\)
- \(g xy = e\) is the same as \(g x = \lambda y \rightarrow e\)
- \(f x y = e\) is the same as \(f = \lambda x y \rightarrow e\)

We say that plusc is in Curried Form; that is, it takes its arguments one at a time.

Consider the following function:

\(\text{curry } f x y = f(x,y)\)

In Haskell we have the following.

```
Main> :t curry
    curry :: ((a,b) -> c) -> a -> b -> c
```
So curry takes a function of type ((a,b) -> c) and returns a function of type (a -> b -> c).

Problem 2.1. Write the function

uncurry :: (a -> b -> c) -> (a,b) -> c.

Problem 2.2.

You may prove the following, or (if you don't want to do the proofs), run extensive tests to show the following hold.

i.) uncurry plusc = plus
ii.) curry plus = plusc
iii.) curry(uncurry plusc) = plusc
iv.) uncurry(curry plus) = plus