Literate Haskell

This is a *literate Haskell* file. Literate Haskell files end in `.lhs` instead of `.hs`, and they switch the usual roles of code and comments: normally, code is the default and comments have to be marked with special syntax; in literate Haskell files, comments are the default and code has to be marked with special syntax. They are useful for creating documents that tell a story and explain some code. There are tools for nicely typesetting literate Haskell documents: for example, I wrote this document using Markdown syntax, and then used the `pandoc` document format conversion tool to turn it into a nicely formatted PDF with syntax highlighted Haskell code, using the command

```
pandoc -t latex RecursionPatterns.lhs -o recursion-patterns.pdf
```

(`pandoc` just so happens to be written in Haskell, though that is a coincidence; it is widely used for non-Haskell things as well.) You can download the compiled PDF from the course website.

Getting started

First, let's turn on a few useful language extensions:

```
{-# LANGUAGE GADTSyntax     #-}
{-# LANGUAGE ExplicitForAll #-}
```

As you can see, code is marked by lines beginning with a `>` sign. Note that (1) there must be a space after the `>`, and (2) there must be blank lines separating blocks of code from comments.

```
x :: Int
x = 3
```

Ghci knows how to parse `.lhs` files in addition to `.hs` files. Make sure you can load this file into ghci and evaluate `x` to ensure it loaded properly.

Warmup: polymorphism

Recall our discussion of parametric polymorphism from last class. Here are a few more function type signatures. For each type signature, write down an implementation of a function that is accepted by GHC, and give a brief explanation of what the function does. Beware! One of them is actually impossible.

```
m :: (a -> a -> b) -> a -> b
m = undefined    -- replace this line with a definition of m

z :: a -> a -> c
z = undefined    -- replace this line with a definition of z
```

```haskell
q :: (b -> c) -> (a -> b) -> a -> c
q = undefined    -- replace this line with a definition of q

s :: (a,a) -> (a,a)
s = undefined    -- replace this line with a definition of s
```

Recursion patterns on lists

At this point in the course you've written a bunch of recursive functions and you know that Haskell fundamentally doesn't have any loops or control structures other than recursion. It may therefore surprise you to learn that experienced Haskell programmers *rarely write* explicitly recursive functions. Instead, there are standard library functions which encapsulate many common *recursion patterns*. Instead of writing a recursive function, an experienced Haskell programmer will call one or more appropriate recursive library functions, which perform the recursion for them.

In this module, you'll explore the implementation of various standard recursion patterns on lists.

For each type signature below:

1. Implement a (recursive) function which has the given type, using parametric polymorphism as a guide.
2. In one or two sentences, explain what your function does.
3. Rename the function to something more evocative of its purpose.
4. Give an example of its use.

In many cases, there are multiple possible ways to implement a function of the given type; you should try to come up with the implementation which feels parsimonious—*i.e.* one which uses all the given inputs to the function in a nontrivial way and which doesn't (needlessly) duplicate or destroy any information.

```haskell
ex1 :: (a -> b) -> [a] -> [b]
ex1 = undefined

ex2 :: (a -> a) -> a -> [a]    -- hint: Haskell is cool with infinite lists
ex2 = undefined

ex3 :: (a -> b -> c) -> [a] -> [b] -> [c]
ex3 = undefined

ex4 :: (a -> Bool) -> [a] -> [a]
ex4 = undefined

ex5 :: (a -> b -> b) -> b -> [a] -> b
ex5 = undefined
```

Applying recursion patterns

Write a Haskell function to accomplish each of the tasks below, using the recursion patterns you implemented in the previous section. Feel free to implement helper functions as necessary. However, *none* of the functions you write in this section are allowed to be recursive; the only recursion allowed is that in the recursion pattern functions from the previous section.

1. Given a list of `Integer`s as input, keep only those which are less than 100, and add 3 to each. Return a list of the results.
2. Given an `Integer` n, construct an (infinite) list of all the powers of n.
3. Compute the dot product of two vectors, represented as lists of `Integer`s. The dot product of $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$ is $x_1y_1 + x_2y_2 + \cdots + x_ny_n$. (You may assume the two input lists have the same length.)

Exploring the standard library

Now visit Hoogle at `haskell.org/hoogle/`, which allows you to search for things in standard Haskell libraries. One cool feature is that in addition to searching by name, Hoogle also allows searching *by type*. Search for each of the five types of your recursion patterns and see if you can find a corresponding function in the standard library. Are your implementations equivalent?