Activity 7

We work with pattern matching a lot when writing functions that deal with lists.
Consider the function to compute the length of a list:

```
len :: [a] -> Int
len [] = 0
len (x:xs) = len xs + 1
```

The pattern [] indicates an empty list and the pattern (x:xs) indicates a list with the first value being denoted by the variable x and everything but the first element denoted by the variable xs.

What pattern would you use for a list containing one element?

Complete the function below which returns the last element in a non-empty list.

```
last :: [a] -> a
last _________ = _______________   -- the case for a list with one element
last (x:xs) = ____________________  -- the case for a list with more than one element
```

Use example in the terminal:

```
> last [1,2,3,4]
4
```

Suppose we have a list of pairs like:
[(1,”one”), (2,”two”),(3,”three”),(4,”four”)]

We could view this as an “association list”, where we are associating a String with each Integer value. A useful function, therefore, would be a lookup function with which we give an Integer and it returns the associated String.

Since we know that we are dealing with a list of pairs, we can use the pattern ((x,y): xs) to indicate that we have a non-empty list whose first value is the pair (x,y) , with x and y as variables for the first and second entry in the pair, and the rest of the list is denoted by the variable xs. In the example above, x would have the value 1, y would have the value “one”, and xs would be the list [(2,”two”),(3,”three”),(4,”four”)].

Complete the function lookup which returns the empty string if the value is not found, and returns the second item in the pair if the value is found:

```
lookup :: Integer -> [(Integer,String)] -> String
lookup n [] = ""
looiup n ((x,y) : xs) = if n == x then
                                  _____________________
                        else
                                  _____________________
```

Use example in the terminal:

```
> let a1 = [(1,"one"), (2,"two"),(3,"three"),(4,"four")]
> lookup 3 a1
"three"
> lookup 5 a1
""
```

Suppose we have a list of Strings and we know all the String values represent numbers like "12" or "-5". Also suppose the we want to sum up the numbers that these String values represent.

One way to do this would be to use the *map* function which is the list equivalent of *every* that we used with sentences. It applies a function to each element of the list.

The function *read* can be used to take a String and convert it to an Integer.

```
stringToInt :: String -> Integer
stringToInt s = read s

> stringToInt "12"
12
> stringToInt "-5"
-5
```

Try using map to change all the Strings to their Integer values:

```
> map stringToInt ["12", "3", "-5"]
```

We could use the *foldl* function, which is the list equivalent of the *accumulate* function that we used with sentences, to add up all the Integer values in a list:

```
> foldl (+) 0 [12,3,-5]
10
```

Combine *foldl* and *map* to define a function which sums the Strings in a list:

```
sumStr :: {String] -> Integer
sumStr x = _____________________________________________________
```

Use example in the terminal

```
> sumStr [“12”, “3”, “-5”]
10
```

What if we never use stringToInt again and just want to define it on the fly for our one time use. Haskell allows us to define “anonymous functions” which are functions that we use once and don’t name. For example,

```
\x -> x + 1
```

is an anonymous function which takes a parameter x and returns x + 1.

We could use this in a map, for example:

```
> map (\x->x+1) [1,2,3]
[2,3,4]
```

Rewrite *sumStr* so that it uses an anonymous function instead of using stringToInt.