```
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

First two patterns say that if the first list or second list is empty, we get an empty list. The third one says that two lists zipped are equal to pairing up their heads and then tacking on the zipped tails. Zipping `[1,2,3]` and `['a','b']` will eventually try to zip `[3]` with `[]`. The edge condition patterns kick in and so the result is `(1,'a'):(2,'b'):[]`, which is exactly the same as `[(1,'a'),(2,'b')]`.

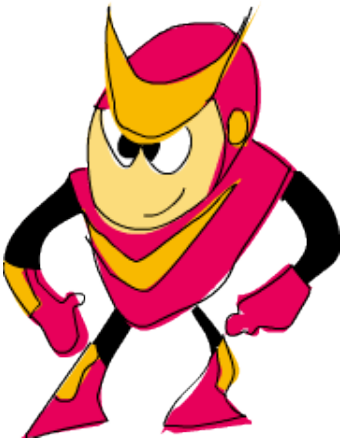
Let's implement one more standard library function — `elem`. It takes an element and a list and sees if that element is in the list. The edge condition, as is most of the times with lists, is the empty list. We know that an empty list contains no elements, so it certainly doesn't have the droids we're looking for.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

Pretty simple and expected. If the head isn't the element then we check the tail. If we reach an empty list, the result is `False`.

Quick, sort!

We have a list of items that can be sorted. Their type is an instance of the `Ord` typeclass. And now, we want to sort them! There's a very cool algoritm for sorting called quicksort. It's a very clever way of sorting items. While it takes upwards of 10 lines to implement quicksort in imperative languages, the implementation is much shorter and elegant in Haskell. Quicksort has become a sort of poster child for Haskell. Therefore, let's implement it here, even though implementing quicksort in Haskell is considered really cheesy because everyone does it to showcase how elegant Haskell is.



So, the type signature is going to be `quicksort :: (Ord a) => [a] -> [a]`. No surprises there. The edge condition? Empty list, as is expected. A sorted empty list is an empty list. Now here comes the main algorithm: **a sorted list is a list that has all the values smaller than (or equal to) the head of the list in front (and those values are sorted), then comes the head of the list in the middle and then come all the values that are bigger than the head (they're also sorted).** Notice that we said *sorted* two times in this definition, so we'll probably have to make the recursive call twice! Also notice that we defined it using the verb *is* to define the algorithm instead of saying *do this, do that, then do that ...*. That's the beauty of functional programming! How are we going to filter the list so that we get only the elements smaller than the head of our list and only elements that are bigger? List comprehensions. So, let's dive in and define this function.

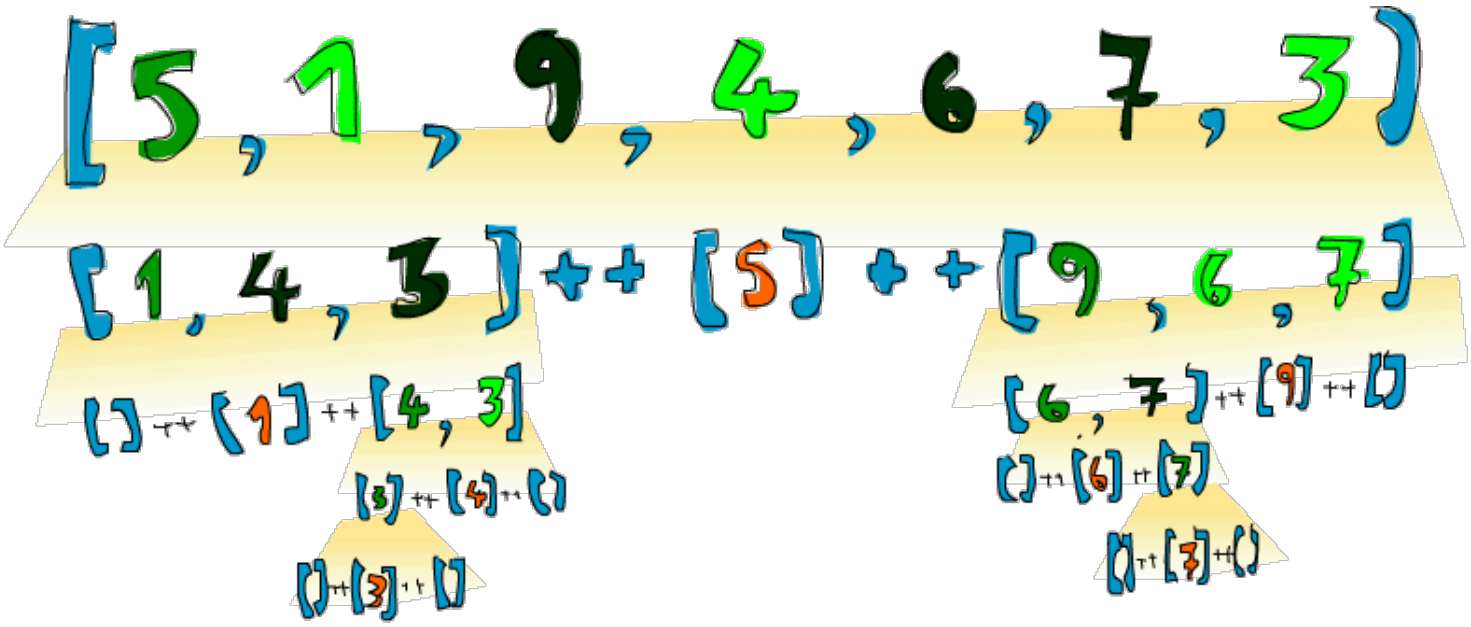
```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
```

```
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Let's give it a small test run to see if it appears to behave correctly.

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

Booyah! That's what I'm talking about! So if we have, say `[5,1,9,4,6,7,3]` and we want to sort it, this algorithm will first take the head, which is `5` and then put it in the middle of two lists that are smaller and bigger than it. So at one point, you'll have `[1,4,3] ++ [5] ++ [9,6,7]`. We know that once the list is sorted completely, the number `5` will stay in the fourth place since there are 3 numbers lower than it and 3 numbers higher than it. Now, if we sort `[1,4,3]` and `[9,6,7]`, we have a sorted list! We sort the two lists using the same function. Eventually, we'll break it up so much that we reach empty lists and an empty list is already sorted in a way, by virtue of being empty. Here's an illustration:



An element that is in place and won't move anymore is represented in **orange**. If you read them from left to right, you'll see the sorted list. Although we chose to compare all the elements to the heads, we could have used any element to compare against. In quicksort, an element that you compare against is called a pivot. They're in **green** here. We chose the head because it's easy to get by pattern matching. The elements that are smaller than the pivot are **light green** and elements larger than the pivot are **dark green**. The yellowish gradient thing represents an application of quicksort.

Thinking recursively

We did quite a bit of recursion so far and as you've probably noticed, there's a pattern here. Usually you define an edge case and then you define a function that does something between some element and the function applied to the rest. It doesn't matter if it's