List Comprehensions

List comprehensions are a really neat way of making a specific list without using functions like `filter` or `take`. If you have done set theory you will notice that the syntax is almost identical (just Haskellified). If you haven't done any set theory then that's okay neither had I. Instead you can of them like wee boxes that you put stuff into.

```
weeBox = [ x | x <- [10..20], x /= 13]
```

There are three main sections of a basic list comprehension. The first section is the box, this is the LHS of the bar or `x` in our example. The next part is what you are putting into the box. You can see that there is a backwards arrow drawing values from `[10..20]` into the `x`. If you didn't have anything else in the list comprehension then the list would just be numbers 10-20, not very interesting. The power of list comprehensions begins to show itself with the next feature. You can put conditions on what goes into the box. Our `weeBox` is currently not letting the number `13` into it because it is superstitious.

```
weeBox = [10,11,12,14,15,16,17,18,19,20]
```

List comprehensions are not limited to one variable on the LHS of the bar. With list comprehensions, there are so many possibilities and the best way to see what they can do this dabble and experiment. List comprehensions are one of the reasons Haskell is so good with manipulation of lists, and allow you to do things that would be very difficult in certain other languages *cough* C *cough* with relative ease.

For example, if you were a particular kind of nerd with a misspent youth you may want to make a function that spits out all possible combinations of two people's names to help you think up ship names. This may even be something that you have wanted to do for a while but didn't even know where to start only using the imperative languages that you had been exposed to. So barely daring to hope that Haskell's list comprehensions could be your salvation, and being the same nerd that plays Quidditch you tentatively experiment by first making a function that will output the hybrid Hogwarts houses:

```
> houses :: [String]
> houses = [start ++ end | start <- start, end <- end]
>    where
>        start  =  ["Slyther", "Huffle", "Raven", "Gryffin"]
>        end    =  ["in", "puff", "claw", "dor"]
```

Then after that works you gain confidence and smash out the function of your dreams. I don't know, maybe I'm just projecting. See if you can work out how `houses` and `ship` work. NOTE:- `tails` and `inits` are functions that are NOT included in the Prelude, they have been imported from the `Data.List` library.

```
> ship :: String -> String -> [String]
> ship a b = [start ++ end | start <- startA, end <- endB] ++ [start ++ end | start <- startB, en
>    where
>        startA  =  [x | x <- inits a, x  /=  []]
>        endB    =  [x | x <- tails b, x  /=  []]
>        startB  =  [x | x <- inits b, x  /=  []]
>        endA    =  [x | x <- tails a, x  /=  []]
```