LIST COMPREHENSIONS

List comprehensions are ways to build or modify lists. They also make programs short and easy to understand compared to other ways of manipulating lists. It's based off the idea of set notation; if you've ever taken mathematics classes with set theory or if you've ever looked at mathematical notation, you probably know how that works. Set notation basically tells you how to build a set by specifying properties its members must satisfy. List comprehensions may be hard to grasp at first, but they're worth the effort. They make code cleaner and shorter, so don't hesitate to try and type in the examples until you understand them!

An example of set notation would be $\{x \in \mathbf{R} : x = x^2\}$. That set notation tells you the results you want will be all real numbers who are equal to their own square. The result of that set would be *{0,1}*. Another set notation example, simpler and abbreviated would be `{x : x > 0}`. Here, what we want is all numbers where *x* > 0.

List comprehensions in Erlang are about building sets from other sets. Given the set `{2n : n in L}` where L is the list *[1,2,3,4]*, the Erlang implementation would be:

```
1> [2*N || N <- [1,2,3,4]].
[2,4,6,8]
```

Compare the mathematical notation to the Erlang one and there's not a lot that changes: brackets ({}) become square brackets ([]), the colon (:) becomes two pipes (||) and the word 'in' becomes the arrow (<-). We only change symbols and keep the same logic. In the example above, each value of *[1,2,3,4]* is sequentially pattern matched to *N*. The arrow acts exactly like the `=` operator, with the exception that it doesn't throw exceptions.

You can also add constraints to a list comprehension by using operations that return boolean values. if we wanted all the even numbers from one to ten, we could write something like:

```
2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 =:= 0].
[2,4,6,8,10]
```

Where `X rem 2 =:= 0` checks if a number is even. Practical applications come when we decide we want to apply a function to each element of a list, forcing it to respect constraints, etc. As an example, say we own a restaurant. A customer enters, sees our menu and asks if he could have the prices of all the items costing between $3 and $10 with taxes (say 7%) counted in afterwards.

```
3> RestaurantMenu = [{steak, 5.99}, {beer, 3.99},
{poutine, 3.50}, {kitten, 20.99}, {water, 0.00}].
[{steak,5.99},
```

```
{beer,3.99},
{poutine,3.5},
{kitten,20.99},
{water,0.0}]
4> [{Item, Price*1.07} || {Item, Price} <- RestaurantMenu,
Price >= 3, Price =< 10].
[{steak,6.409300000000001},{beer,4.2693},{poutine,3.745}]
```

Of course, the decimals aren't rounded in a readable manner, but you get the point. The recipe for list comprehensions in Erlang is therefore `NewList = [Expression || Pattern <- List, Condition1, Condition2, ... ConditionN]`. The part `Pattern <- List` is named a Generator expression. You can have more than one!

```
5> [X+Y || X <- [1,2], Y <- [2,3]].
[3,4,4,5]
```

This runs the operations `1+2`, `1+3`, `2+2`, `2+3`. So if you want to make the list comprehension recipe more generic, you get: `NewList = [Expression || GeneratorExp1, GeneratorExp2, ..., GeneratorExpN, Condition1, Condition2, ... ConditionM]`. Note that the generator expressions coupled with pattern matching also act as a filter:

```
6> Weather = [{toronto, rain}, {montreal, storms},
{london, fog},
6>     {paris, sun}, {boston, fog}, {vancouver, snow}].
[{toronto,rain},
{montreal,storms},
{london,fog},
{paris,sun},
{boston,fog},
{vancouver,snow}]
7> FoggyPlaces = [X || {X, fog} <- Weather].
[london,boston]
```

If an element of the list 'Weather' doesn't match the {X, fog} pattern, it's simply ignored in the list comprehension whereas the `=` operator would have thrown an exception.

There is one more basic data type left for us to see for now. It is a surprising feature that makes interpreting binary data easy as pie.

Source : http://learnyousomeerlang.com/starting-out-for-real#numbers