LISTS IN ERLANG

Lists are the bread and butter of many functional languages. They're used to solve all kinds of problems and are undoubtedly the most used data structure in Erlang. Lists can contain anything! Numbers, atoms, tuples, other lists; your wildest dreams in a single structure. The basic notation of a list is `[Element1, Element2, ..., ElementN]` and you can mix more than one type of data in it:

```
1> [1, 2, 3, {numbers,[4,5,6]}, 5.34, atom].
[1,2,3,{numbers,[4,5,6]},5.34,atom]
```

Simple enough, right?

```
2> [97, 98, 99].
"abc"
```

Uh oh! This is one of the most disliked things in Erlang: strings! Strings are lists and the notation is absolutely the exact same! Why do people dislike it? Because of this:

```
3> [97,98,99,4,5,6].
[97,98,99,4,5,6]
4> [233].
"é"
```

Erlang will print lists of numbers as numbers only when at least one of them could not also represent a letter! There is no such thing as a real string in Erlang! This will no doubt come to haunt you in the future and you'll hate the language for it. Don't despair, because there are other ways to write strings we'll see later in this chapter.

Don't drink too much Kool-Aid:
This is why you may have heard Erlang is said to suck at string manipulation: there is no built-in string type like in most other languages. This is because of Erlang's origins as a language created and used by telecom companies. They never (or rarely) used strings and as such, never felt like adding them officially. However, most of Erlang's lack of sense in string manipulations is getting fixed with time: The VM now natively supports Unicode strings, and overall gets faster on string manipulations all the time.
There is also a way to store strings as a binary data structure, making them really light and faster to work with. All in all, there are still some functions missing from the standard library and while string processing is definitely doable in Erlang, there are somewhat better languages for tasks that need lots of it, like Perl or Python.

To glue lists together, we use the `++` operator. The opposite of `++` is `--` and will remove elements from a list:

```
5> [1,2,3] ++ [4,5].
[1,2,3,4,5]
```

```
6> [1,2,3,4,5] -- [1,2,3].
[4,5]
7> [2,4,2] -- [2,4].
[2]
8> [2,4,2] -- [2,4,2].
[]
```

Both `++` and `--` are right-associative. This means the elements of many `--` or `++` operations will be done from right to left, as in the following examples:

```
9> [1,2,3] -- [1,2] -- [3].
[3]
10> [1,2,3] -- [1,2] -- [2].
[2,3]
```

Let's keep going. The first element of a list is named the Head, and the rest of the list is named the Tail. We will use two built-in functions (BIF) to get them.

```
11> hd([1,2,3,4]).
1
12> tl([1,2,3,4]).
[2,3,4]
```

Note: built-in functions (BIFs) are usually functions that could not be implemented in pure Erlang, and as such are defined in C, or whichever language Erlang happens to be implemented on (it was Prolog in the 80's). There are still some BIFs that could be done in Erlang but were still implemented in C in order to provide more speed to common operations. One example of this is the `length(List)` function, which will return the (you've guessed it) length of the list passed in as the argument.

Accessing or adding the head is fast and efficient: virtually all applications where you need to deal with lists will always operate on the head first. As it's used so frequently, there is a nicer way to separate the head from the tail of a list with the help of pattern matching: `[Head|Tail]`. Here's how you would add a new head to a list:

```
13> List = [2,3,4].
[2,3,4]
14> NewList = [1|List].
[1,2,3,4]
```

When processing lists, as you usually start with the head, you want a quick way to also store the tail to later operate on it. If you remember the way tuples work and how we used pattern matching to unpack the values of a point (*{X, Y}*), you'll know we can get the first element (the head) sliced off a list in a similar manner.

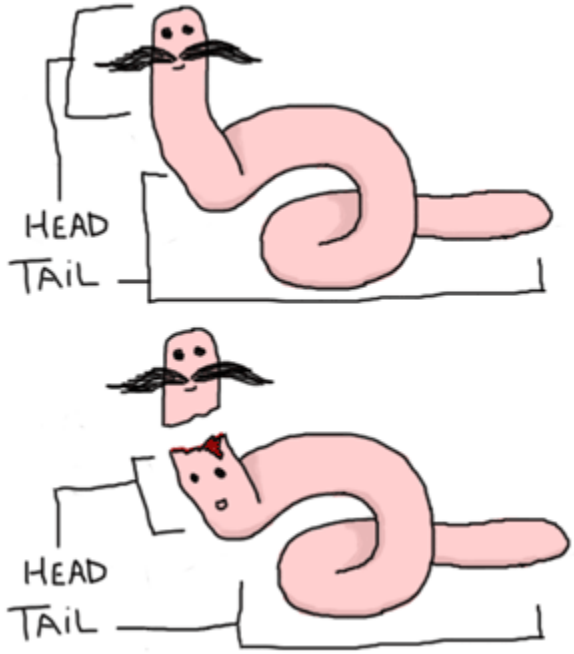
```
15> [Head|Tail] = NewList.
[1,2,3,4]
16> Head.
1
17> Tail.
```

```
[2,3,4]
18> [NewHead|NewTail] = Tail.
[2,3,4]
19> NewHead.
2
```

The `|` we used is named the cons operator (constructor). In fact, any list can be built with only cons and values:

```
20> [1 | []].
[1]
21> [2 | [1 | []]].
[2,1]
22> [3 | [2 | [1 | [] ] ].
[3,2,1]
```

This is to say any list can be built with the following formula: `[Term1| [Term2 | [... | [TermN]]]]...`. Lists can thus be defined recursively as a head preceding a tail, which is itself a head followed by more heads. In this sense we could imagine a list being a bit like an earthworm: you can slice it in half and you'll then have two worms.



The ways Erlang lists can be built are sometimes confusing to people who are not used to similar constructors. To help you get familiar with the concept, read all of these examples (hint: they're all equivalent):

```
[a, b, c, d]
[a, b, c, d | []]
[a, b | [c, d]]
[a, b | [c | [d]]]
[a | [b | [c | [d]]]]
[a | [b | [c | [d | [] ]]]]
```

With this understood, you should be able to deal with list comprehensions.

Note: Using the form `[1 | 2]` gives what we call an 'improper list'. Improper lists will work when you pattern match in the `[Head|Tail]` manner, but will fail to be used with standard functions of Erlang (even `length()`). This is because Erlang expects proper lists. Proper lists end with an empty list as their last cell. When declaring an item like `[2]`, the list is automatically formed in a proper manner. As such, `[1|[2]]` would work! Improper lists, although syntactically valid, are of very limited use outside of user-defined data structures.

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