HIGHER ORDER FUNCTIONS IN ERLANG

Let's get functional

An important part of all functional programming languages is the ability to take a function you defined and then pass it as a parameter to another function. This in turn binds that function parameter to a variable which can be used like any other variable within the function. A function that can accept other functions transported around that way is named a*higher order function*. Higher order functions are a powerful means of abstraction and one of the best tools to master in Erlang.

Again, this a concept rooted in mathematics, mainly lambda calculus. I won't go into much detail about lambda calculus because some people have a hard time grasping it and it's a bit out of scope. However, I'll define it briefly as a system where everything is a function, even numbers. Because everything is a function, functions must accept other functions as parameters and can operate on them with even more functions!

Alright, this might be a little bit weird, so let's start with an example:

```
-module(hhfuns).
-compile(export_all).

one() -> 1.
two() -> 2.

add(X,Y) -> X() + Y().
```

Now open the Erlang shell, compile the module and get going:

```
1> c(hhfuns).
{ok, hhfuns}
2> hhfuns:add(one,two).
** exception error: bad function one
in function  hhfuns:add/2
3> hhfuns:add(1,2).
** exception error: bad function 1
in function  hhfuns:add/2
4> hhfuns:add(fun hhfuns:one/0, fun hhfuns:two/0).
3
```

Confusing? Not so much, once you know how it works (isn't that always the case?) In command 2, the atoms `one`and `two` are passed to `add/2`, which then uses both atoms as function names (`X() + Y()`). If function names are written without a parameter list then those names are interpreted as atoms, and atoms can not be functions, so the call fails. This is the reason why expression 3 also fails: the values 1 and 2 can not be called as functions either, and functions are what we need!

This is why a new notation has to be added to the language in order to let you pass functions from outside a module. This is what `fun Module:Function/Arity` is: it tells the VM to use that specific function, and then bind it to a variable.

So what are the gains of using functions in that manner? Well a little example might be needed in order to understand it. We'll add a few functions to `hhfuns` that work recursively over a list to add or subtract one from each integer of a list:

```
increment([]) -> [];
increment([H|T]) -> [H+1|increment(T)].

decrement([]) -> [];
decrement([H|T]) -> [H-1|decrement(T)].
```

See how similar these functions are? They basically do the same thing: they cycle through a list, apply a function on each element (`+` or `-`) and then call themselves again. There is almost nothing changing in that code: only the applied function and the recursive call are different. The core of a recursive call on a list like that is always the same. We'll abstract all the similar parts in a single function (`map/2`) that will take another function as an argument:

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].

incr(X) -> X + 1.
decr(X) -> X - 1.
```

Which can then be tested in the shell:

```
1> c(hhfuns).
{ok, hhfuns}
2> L = [1,2,3,4,5].
[1,2,3,4,5]
3> hhfuns:increment(L).
[2,3,4,5,6]
4> hhfuns:decrement(L).
[0,1,2,3,4]
5> hhfuns:map(fun hhfuns:incr/1, L).
[2,3,4,5,6]
6> hhfuns:map(fun hhfuns:decr/1, L).
[0,1,2,3,4]
```

Here the results are the same, but you have just created a very smart abstraction! Every time you will want to apply a function to each element of a list, you only have to call `map/2` with your function as a parameter. However, it is a bit annoying to have to put every function we want to pass as a parameter to `map/2` in a module, name it, export it, then compile it, etc. In fact it's plainly unpractical. What we need are functions that can be declared on the fly...

Source : http://learnyousomeerlang.com/higher-order-functions