PATTERN MATCHING

Now that we have the ability to store and compile our code, we can begin to write more advanced functions. Those that we have written so far were extremely simple and a bit underwhelming. We'll get to more interesting stuff. The first function we'll write will need to greet someone differently according to gender. In most languages you would need to write something similar to this:

```
function greet(Gender,Name)
if Gender == male then
print("Hello, Mr. %s!", Name)
else if Gender == female then
print("Hello, Mrs. %s!", Name)
else
print("Hello, %s!", Name)
end
```

With pattern-matching, Erlang saves you a whole lot of boilerplate code. A similar function in Erlang would look like this:

```
greet(male, Name) ->
io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
io:format("Hello, ~s!", [Name]).
```

I'll admit that the printing function is a lot uglier in Erlang than in many other languages, but that is not the point. The main difference here is that we used pattern matching to define both what parts of a function should be used and bind the values we need at the same time. There was no need to first bind the values and then compare them! So instead of:

```
function(Args)
if X then
Expression
else if Y then
Expression
else
Expression
```

We write:

```
function(X) ->
Expression;
function(Y) ->
Expression;
function(_) ->
Expression.
```

in order to get similar results, but in a much more declarative style. Each of these `function` declarations is called a*function clause*. Function clauses must be separated by semicolons (`;`) and together form a *function declaration*. A function declaration counts as one larger statement, and it's why the final function clause ends with a period. It's a "funny" use of tokens to determine workflow, but you'll get used to it. At least you'd better hope so because there's no way out of it!

Note: `io:format`'s formatting is done with the help of tokens being replaced in a string. The character used to denote a token is the tilde (`~`). Some tokens are built-in such as `~n`, which will be changed to a line-break. Most other tokens denote a way to format data. The function call`io:format("~s!~n",["Hello"]).` includes the token `~s`, which accepts strings and bitstrings as arguments, and `~n`. The final output message would thus be `"Hello!\n"`. Another widely used token is `~p`, which will print an Erlang term in a nice way (adding in indentation and everything).
The `io:format` function will be seen in more details in later chapters dealing with input/output with more depth, but in the meantime you can try the following calls to see what they do:`io:format("~s~n",[<<"Hello">>])`, `io:format("~p~n",[<<"Hello">>])`, `io:format("~~~n")`,`io:format("~f~n", [4.0])`, `io:format("~30f~n", [4.0])`. They're a small part of all that's possible and all in all they look a bit like `printf` in many other languages. If you can't wait until the chapter about I/O, you can read the online documentation to know more.

Pattern matching in functions can get more complex and powerful than that. As you may or may not remember from a few chapters ago, we can pattern match on lists to get the heads and tails. Let's do this! Start a new module called `functions` in which we'll write a bunch of functions to explore many pattern matching avenues available to us:

```
-module(functions).
-compile(export_all).%% replace with -export() later, for
God's sake!
```

The first function we'll write is `head/1`, acting exactly like `erlang:hd/1` which takes a list as an argument and returns its first element. It'll be done with the help of the cons operator (`|`):

```
head([H|_]) ->H.
```

If you type `functions:head([1,2,3,4]).` in the shell (once the module is compiled), you can expect the value '1' to be given back to you. Consequently, to get the second element of a list you would create the function:

```
second([_,X|_]) ->X.
```

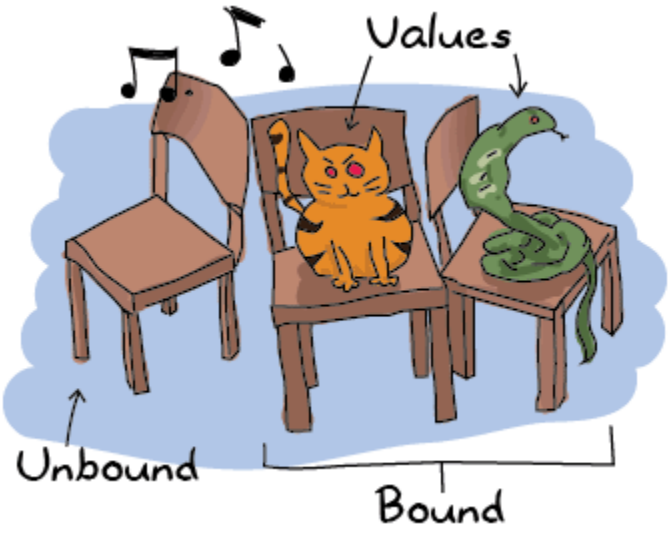
The list will just be deconstructed by Erlang in order to be pattern matched. Try it in the shell!

```
1> c(functions).
{ok, functions}
2> functions:head([1,2,3,4]).
1
3> functions:second([1,2,3,4]).
2
```

This could be repeated for lists as long as you want, although it would be impractical to do it up to thousands of values. This can be fixed by writing recursive functions, which we'll see how to do later on. For now, let's concentrate on more pattern matching. The concept of free and bound variables we discussed in Starting Out (for real) still holds true for functions: we can then compare and know if two parameters passed to a function are the same or not. For this, we'll create a function `same/2` that takes two arguments and tells if they're identical:

```
same(X,X) ->
true;
same(_,_) ->
false.
```

And it's that simple. Before explaining how the function works, we'll go over the concept of bound and unbound variables again, just in case:



If this game of musical chairs was Erlang, you'd want to sit on the empty chair. Sitting on one already occupied wouldn't end well! Joking aside, unbound variables are variables without any values attached to them (like our empty chair). Binding a variable is simply attaching a value to an unbound variable. In the case of Erlang, when you want to assign a value to a variable that is already bound, an error occurs *unless the new value is the same as the old one*. Let's imagine our snake on the right: if another snake comes around, it won't really change much to the game. You'll just have more angry snakes. If a different animal comes to sit on the chair (a honey badger, for example), things will go bad. Same values for a bound variable are fine, different ones are a bad idea. You can go back to the subchapter about Invariable Variables if this concept is not clear to you.

Back to our code: what happens when you call `same(a,a)` is that the first *X* is seen as unbound: it automatically takes the value `a`. Then when Erlang goes over to the second argument, it sees *X* is already bound. It then compares it to the `a` passed as the second argument and looks to see if it matches. The pattern matching succeeds and the function returns `true`. If the two values aren't the same, this will fail and go to the second function clause, which doesn't care about its arguments (when you're the last to choose, you can't be picky!) and will instead return false. Note that this function can effectively take any kind of argument whatsoever! It works for any type of data, not just lists or single variables. As a rather advanced example, the following function prints a date, but only if it is formatted correctly:

```
valid_time({Date = {Y,M,D}, Time = {H,Min,S}}) ->
io:format("The Date tuple (~p) says today is:
~p/~p/~p,~n",[Date,Y,M,D]),
io:format("The time tuple (~p) indicates: ~p:~p:~p.~n",
[Time,H,Min,S]);
valid_time(_) ->
io:format("Stop feeding me wrong data!~n").
```

Note that it is possible to use the `=` operator in the function head, allowing us to match both the content inside a tuple (`{Y,M,D}`) and the tuple as a whole (*`Date`*). The function can be tested the following way:

```
4> c(functions).
{ok, functions}
5> functions:valid_time({{2011,09,06},{09,04,43}}).
The Date tuple ({2011,9,6}) says today is: 2011/9/6,
The time tuple ({9,4,43}) indicates: 9:4:43.
ok
6> functions:valid_time({{2011,09,06},{09,04}}).
Stop feeding me wrong data!
ok
```

There is a problem though! This function could take anything for values, even text or atoms, as long as the tuples are of the form `{{A,B,C}, {D,E,F}}`. This denotes one of the limits of pattern matching: it can either specify really precise values such as a known number of atom, or abstract values such as the head|tail of a list, a tuple of *N*elements, or anything (`_` and unbound variables), etc. To solve this problem, we use guards.

Source : http://learnyousomeerlang.com/syntax-in-functions