# Functional Programming in Erlang



The aim of these exercises is to consolidate your understanding and use of functions that take functions as arguments and return functions as results. We'll look at function composition and iteration.

Do try the first two exercises - *In the shell* and *Composition* - and if you want more practice or to stretch yourself you could also have a go at the further exercises *Twice* and *Iteration*. Use the comments for this step to share your questions, comments and solutions for these exercises, and to help provide feedback for others.

There is a supporting file, `hof.erl`, available under 'Downloads' below, to help you get started.

---

## In the shell

Try defining functions for yourself in the shell. You can do this using `fun` expressions, such as:

```
Add = fun (X,Y) -> X+Y end.

Sum = fun (Xs) -> lists:foldr(Add,0,Xs) end.
```

`fun` expressions can also use pattern matching, as in:

```
EmptyTest = fun ([]) -> true ; ([_|_]) -> false end.
```

Once these functions are defined you can apply them in the usual way, as in:

```
28> Add(1,2).
3
29> Sum([3,4,67]).
74
30> EmptyTest([3,4,67]).
false
```

Finally, `fun` expressions can be *recursive*:

```
31> Foo = fun Product([]) -> 1 ; Product([X|Xs]) -> X*Product(Xs) end.
#Fun<erl_eval.30.90072148>
32> Foo([0,1,2]).
0
33> Foo([10,1,2]).
20
34> Product([10,1,2]).
* 1: variable 'Product' is unbound
```

---

## Composition

Composition is *associative* – meaning that it doesn't matter how you bracket a composition of three functions – but it's not *commutative* – so that `F` composed with `G` is not necessarily the same as `G` composed with `F`. Find examples

of `F` and `G` to show cases when the two compositions are the same, and when they are different.

Define a function that takes a *list* of functions and composes them together. Can you use any of the functions in the `lists` module to help you?

---

### Twice

Using `compose` or otherwise, define a function `twice` that applies a function to an argument twice. For example, applying “multiply by 3” twice to the argument 2 gives 18 (as applying it once gives 6 and then applying it again gives 18).

What happens when you apply `twice` to itself? What happens when you apply the result of that to “multiply by 3” and the result of that to 2?

---

### Iteration

Define a function `iterate` that takes a number `N` and returns a function that takes a function and returns that function iterated `N` times. When `N` is zero, it should return the identity function (that is, the function that returns its argument unchanged).

---