# Higher Order Functions

Aditya Gupta

February 2023

## 1 Introduction

Higher-order functions are functions that can take other functions as arguments, where the argument function can be applied on the other arguments for the composing function.

## 2 Reducing a List

Imagine if we have a list of values that we want to operate on, for example if we want to find the sum of all the elements or to find the average or if we wanted to find the minimum or the maximum value in the list.
One way of solving this would be to create 4 separate functions for each operation we want to perform on the list. But now using the aid of Higher Order functions we can just write one function that can do all of those things depending on what argument is passes to it. Using higher order functions we can just pass the functionality we want as an argument, which makes our code a lot shorter and a lot more versatile. The following code is used for finding the sum and the minimum value for a list using separate functions for each of the operations.

```
def sum([], acc) do acc end
def sum([head|tail],acc) do
    sum(tail,head+acc)
end

def minM([], min) do end
def minM([head|tail], min) do
    cond do
        head < min -> minM(tail,min)
        true -> minM(tail,min)
end
```

The same thing that we do above can be done using just 2 lines of code if we make use of Higher-Order functions,in the following way:-

```
Enum.reduce(list, 0, fn(x, acc) -> x + acc end)
# To obtain the sum of all elements in a list
Enum.reduce(list, fn(x,acc) -> cond do x < acc -> x; true -> acc; end end)
#To find the smallest element in a list
```

Here we use an in-built Elixir function Enum.reduce/3 which takes an enumerable data structure, a accumulator and a function. The function is applied to each element in the argument data structure then the return value is stored in the accumulator. This hides a lot that happens from us which might make it harder to understand but we can also implement our own implementation for the reduce function as well in the following way:

```
def reduce([],acc,_) do acc end
def reduce([head|tail],acc,f) do
    reduce(tail, f.(head,acc), f)
end
# This is a bit easier to understand compared to Enum.reduce/3.
```

An example of using the above function can be :-

```
reduce([1,2,3], 1, fn(x, acc) -> x * acc end)
# This function results in the value 6 i.e. the product of all
# values in the list.
```

## 3 Syntax for using Functions as arguments in Higher-Order Functions

When we pass functions as arguments, we can only pass special functions called lambda functions as arguments using the *fn* keyword instead of just normal functions declared using the *def* keyword. We declare a lambda function in the following way:

```
f = fn(x) -> x * 2 end
# Here we can store a function that doubles every element passed
# to it inside a variable f.
g = fn(x) -> IO.puts("Value: #{f.(x)}") end

# The functions created above can be used as arguments
IO.puts(Enum.each([34,65,24,69,210,g))
```

We get the following result upon executing the above code:

```
iex(3)> f = fn(x) -> x * 2 end
#Function<42.3316493/1 in :erl_eval.expr/6>
iex(4)> g = fn(x) -> IO.puts("Value: #{f.(x)}") end
#Function<42.3316493/1 in :erl_eval.expr/6>
iex(5)> IO.puts(Enum.each([34,65,24,69,210],g))
Value: 68
Value: 130
Value: 48
Value: 138
Value: 420
ok
:ok
iex(6)>
```

## 4 Applications

We can use Higher order functions for a variety of purposes including filtering elements from a list based on a specific criteria, or for reducing a list to just a value. Some of the most common higher order functions from the Enum module in Elixir:

- Enum.map/2 → This function is used to apply a lambda function on all elements inside of a list. Example:

  ```
  Enum.map([8,16,32], fn(x) -> x * 3 end) # Result: [24,48,96]
  ```

- Enum.reduce/3 → This is used to accumulate a result from the argument list and lambda function provided by the user. Example:

  ```
  Enum.reduce([4,3,2,1],1,fn(x,acc) -> x * acc end) # Result: 24
  ```

- Enum.filter/2 → Used to filter functions that match against a certain criteria. Example:

  ```
  Enum.filter([0,-1,3,5,7], fn(x) -> rem(x,2) == 1 end) #Result: [3, 5, 7]
  ```

- Enum.each/2 → Invokes the function on each element in the list provided as an argument. Example:

  ```
  Enum.each(["Hello", "World"], fn x -> IO.puts(x) end)
  #Result:
  #"Hello"
  #"World"
  ```

We can see that Higher-Order function are quite versatile as we can pass different function to them to get different functionality. Using higher order functions also helps make functions more generic and be applicable for different kinds of data structures and different data types.