Higher order functions

Björn Formgren

Spring 2023

Introduction

The second higher grade assignment in ID1019 consisted of implementing two constructs: Map and Reduce.

Double, Five and Animals

The first part of the assignment wewre to write a couple of short programs that took a list of a certain data structure and changed it in some way. The three functions were implemented like so:

```
def double ([]) do [] end
def double ([h|t]) do [h * 2|double(t)] end

def five([]) do [] end
def five([h|t]) do [h + 5|five(t)] end

def animal([]) do [] end
def animal([:dog|t]) do [:fido | animal(t)] end
def animal([h|t]) do [h|animal(t)] end
```

The point was to notice that all three functions basically do the same thing, we can generalize this.

```
def double_five_animal([], _) do [] end
def double_five_animal([h|t], op) do
  case op do
    :double -> [h * 2|double_five_animal(t,:double)]
    :five -> [h + 5|double_five_animal(t, :five)]
    :animal ->
      case h do
      :dog  -> [:fido | double_five_animal(t, :animal)]
      _     -> [h|double_five_animal(t, :animal)]
    end
  end
end
```

There exists another way of solving this in elixir, using anonymous functions. Since functions can be assigned to variables in elixir, we can define a function *apply to all* that takes a list and a function. It should apply the function on every element in the list. If the functions *f, g, h* are defined as our double, five and animal functions we can then call the apply functions on their respective lists with the corresponding function.

```
def apply_to_all([], _) do [] end
def apply_to_all([h|t], f) do [f.(h)|apply_to_all(t, f)] end

iex(33)> Higher.apply_to_all([1,2,3,4],f)
[2, 4, 6, 8]
```

Fold

In the next assignment we wanted to sum up all the elements in list using the techniques above. We implemented two functions *fold right/left*. These two functions take three arguments, a list, a base value and a function. The purpose of the function was to apply the argument function to every element in the list as before. We could then implement a sum and product function and assign them to variables to use as an argument in the fold functions. Since addition and multiplication are both symmetric functions it does not matter in which order we pass the arguments. The fold left function was implemented as tail recursive, as per it's definition this means that the last statement to be executed is the recursive call which means that we can reuse the stack frame.

```
def fold_right([], val, _) do val end
def fold_right([h|t], val, f) do f.
     (h,fold_right(t,val,f)) end

def fold_left([], acc, _) do acc end
def fold_left([h|t], acc, f) do
    fold_left(t,f.(h,acc), f)
```

Filter

```
iex(85)> i = fn(x) -> rem(x,2) == 1 end
iex(92)> i = fn(x) -> rem(x,2) == 0 end
iex(98)> i = fn(x) -> x > 5 end

 def filter([], _) do [] end
  def filter([h|t], f) do
    if f.(h)
```

```
    do
     [h|filter(t, f)]
    else
     filter(t, f)
   end
 end
```

The last assignment discussed filtering out items in a list given a certain condition. The same method as before was used to create a generalized function *filter*.