Higher order functions

Programming II - Elixir Version

Johan Montelius

Spring Term 2023

Introduction

In this assignment you will learn how to implement two very powerful constructs: map and reduce. Once you master these higher order functions you will cut the source code that you have to write by a factor of two.

Recursively transforming a list

Let's start with a simple example; implement a function that takes a list of integers as input and returns a list where all elements have been doubled.

```
def double( ... ) do .. end
def double( ... ) do
   :
end
```

Now implement a function that will take a list of integers and returns a list where you have added five to each element.

```
def five( ... ) do .. end
def five( ... ) do
   :
end
```

Implement a function, `animal/1`, that takes a list of animals: `:cat`, `:dog`, `:cow` etc and replaces each occurrence of `:dog` with `:fido`.

As you now hopefully see is that your three functions all have the same pattern. We should be able to implement one function that does the recursion for us and performs the transformation that we want to do.

Remove all of your functions and implement a function `double_five_animal/2` that takes a list of either integers or animals and performs a transformation depending on the second argument. If the second argument is `:double` you should double the element, if it is `:five` you should etc.

```
def double_five_animal( ..., ...) do ... end
def double_five_animal( ..., ...) do
   case ...  do
     ...   ->   ...
     ...   ->   ...
     ...   ->   ...
   end
end
```

Now, using your implementation of `double_five_animal/2`, implement the functions `double/1`, `five/1` and `animal`.

Functions as data

We will now introduce something that for ever will change the way you program. In the programming you have done so far you have probably always seen the program and its procedures as one part and the data structures that you work with as another part. This does not have to be the case, if the language allows we can treat functions as data.

In the Elixir shell try the following:

```
f = fn(x) -> x * 2 end
```

What is `f`? The answer is that `f` is a function (or rather a closure but more on this later). It's a function that takes one argument and returns the doubling of this argument. How do you use it, try the following:

```
f.(5)
```

Notice the dot between the `f` and the parenthesis, if `f` was the name of a function we would write `f(5)` but now since it is a variable bound to a function we have to use the dot notation.

In the same way you can construct a function `g` that adds five to its argument and a function `h` that returns `:fido` if applied to `:dog` otherwise returns the argument as is.

`You can use the one row if-then-else: if x == :dog, do: :fido , else: x`

Now for the revolution.

A function as argument

If we no can hold a function in our hand we should of course also be able to pass it as an argument to another function. Let's go back and look at our original problem, implementing a generic transformer that transforms each element in a list.

Implement a function called apply_to_all/2 that takes a list as its first argument and ... a function ... as it's second argument. The function apply_to_all/2 should generate a list that is the result of applying the given function to each of the elements in the list.

```
def apply_to_all(..., ...) do ... end
def apply_to_all(..., ...) do
  :
end
```

Now, give that you have defined f, g and h, try the following in the terminal:

```
apply_to_all([1,2,3,4], f)

apply_to_all([1,2,3,4], g)

apply_to_all([:dog, :cat, :cow, :horse], h)
```

Hmm, what if we try this:

```
apply_to_all([1,2,3,4], fn(x) -> 2*x + 5 end)
```

As you see the function apply_to_all/2 is quite versatile and can be used any time you want to run through a list of items and apply a function to each of its elements. It's so useful that it has been given a very short name map/2 and is part of the Enum module.

Reducing a list

Let's try another scenario, assume that we should sum the elements of a list. This should be something that you now can do in less than a minute (don't us an accumulator or try to do a tail-recursive implementation), give it a try:

```
def sum( ... ) do ... end
def sum( ... ) do
  ...
end
```

As you see, this function also uses a simple recursive pattern, you're running down the list and adding the element to whatever the result is from the recursive call. The result of summing an empty list is of course 0.

If we follow the idea from `apply_to_all/2` we could rewrite this using one generic function `fold_right/3` that takes a list, a base value and a function as arguments. It should of course return the base value for an empty list and otherwise return the value obtained by applying the function to an element and whatever is returned by the recursive call to `fold_right/3`.

```
def fold_right(..., ..., _ ) do ... end
def fold_right(..., ..., f ) do
  f.(..., ...)
end
```

Using `fold_right/2` you can now implement both `sum/1` and `prod/2` (that returns the product of all elements).

You might wonder why it we call it `_right` and the reason is that we apply the function in a right to left order. When we sum the elements of the list `[1,2,3,4]` we will apply the addition as follows:

$$1 + (2 + (3 + (4 + 0)))$$

Since addition is a symmetric function it does not matter if we pass the function `f(x,y) -> x + y end` or `f(y,x) -> x + 4 end` to `fold_right/2` but most functions are not symmetric so we need to agree on how we pass the arguments to the given function. Should we do `f.(elem, fold_right(...))` or `f.(fold_right(...), elem)`? The agreement is to do it as in the first example so if you try the following:

```
fold_right([1,2,3,4], 0, fn(x, acc) -> {x, acc} end)
```

you should receive the answer:

```
{1, {2, {3, {4, 0}}}}
```

Now if there is a `fold_right/3`, obviously there must be a `fold_left/3`. This function will do the same thing but now apply the function left-to-right. If we try:

```
fold_left([1,2,3,4], 0, fn(x, acc) -> {x, acc} end)
```

we should receive:

```
{4, {3, {2, {1, 0}}}}
```

To implement `fold_left/3` you need to use you skills in writing a tail recursive function where we have an accumulating parameter. The second argument is the accumulating parameter so the implementation looks like this:

```
def fold_left(..., ..., _ ) do ... end
def fold_left(..., ..., f ) do
  fold_left(..., f.(..., ...), f)
end
```

The left-to-right version is in most cases more efficient since we are not using any stack space but it depends on the function that apply. With a function with a constant execution time it does not matter if we apply the function left-to-right or right-to-left but if the execution depends on the size of either of its arguments we need to be careful.

The fold functions are so useful that they are provided in the module `List` and they are called `foldl/3` and `foldr/3`. There is also an implementation in the module `Enum` called `reduce/3` that implements the left-to-right strategy.

The reason it also occurs in the `Enum` module is that this module can handle other enumerable data structures, not just lists.

Filter out the good ones

The third higher order construct that we shall look at is one that filters out the elements in a list that meet some requirement. You have now seen several examples of how we can define a higher order function so this one should be no match for you.

Start by implementing a function `odd/1` that takes a list and returns a list of all the odd elements (`rem(x,2) == 1`).

```
def odd( ... ) do ... end
def odd( ... ) do
  if ...  do
     ...
  else
     ...
  end
end
```

Now rewrite this as a function `filter/2` that takes a list and a function that returns `true` or `false`. You should then be able to implement not only the function `odd/1` using `filter/2` but also functions as `even/1` or `greater_than_five/1`.

Summary

Once you start to use the higher order functions `map/2`, `reduce/3` and `filter/2` you will be able to write much shorter programs and spend less time writing recursive functions that looks more or less the same. It is very often that most of a program actually consist of a series of these operations (and their cousins) that once you get use to it your programs will not look the same.

These higher order constructs are available in all functional programming languages, you will find them in Haskell as well as in JavaScript. While the hard core functional programmers argue that the lambda calculus, Church numerals and the Y combinator is the reason to use functional programming, most programmers just finds map, reduce and filter so useful that they need no other reason.

In the future, write your programs as normal but then spend some time to see if you can rephrase it using the higher order functions found in the `Enum` module.