Operations on a list

Ruxandra-Stefania Tudose

Spring Term 2024

Introduction

This report aims to open a window onto the implementation of ten different functions using regular recursion. Before diving into sketching how the Elixir code would look like, as indicated, I grouped the functions into three different patterns they followed.

First pattern: *the length, sum and product of a list*

In order to find out the length of the list, as long as we do not reach its end, we have to access each element and therefore, add a 1 for for every head we come across. As far as the other two functions are concerned, based on the operation, the following changes have to made:

- sum: change the base case result to zero and add the head of every sublist with the function's recursive call;
- prod: change the base case result to 1 (not very mathematical compliant) and multiply each head of the sublist with the recursive step;

As far as the product base case is concerned, the value 1 is not very mathematical compliant since we are doing the multiplication of nothing, however it is needed in order to get the correct final results.

```
def list_length([]) do 0 end
def list_length([h | t]) do 1 + list_length(t) end

def sum([]) do 0 end
def sum([h | t]) do h + sum(t) end

def prod([]) do 1 end
def prod([h | t]) do h * prod(t) end
```

Second pattern: *creating a modified version of the initial list as indicated*

This time, the output will be a list and the methods follow quite a similar approach. Below I have included the function that returns the remainders of the elements given an integer. I have chosen to display a corresponding error atom for the zero division, while the last step is turning the remainder into one element list and appending it to the result of the recursive step. For the other functions, the division by zero will be removed and the operation applied on the list's head will be changed accordingly.

```
def list_rem(_, 0) do :error_division_by_zero end
def list_rem([], _) do [] end
def list_rem([h | t], n) do
  [rem(h,n)] ++ list_rem(t, n)
end
```

Third pattern: *the even, odd and evenly divisible list*

Apart from the standard base cases, these three functions will include in their pattern match, a *case* in order to check the remainder of the division, depending on which, the next code line will be executed. For the *even and odd* functions we have to check whether the remainder is 0 or 1 (for even: if it is a zero, it's an element that satisfies the condition and we add it to the list, otherwise we keep on going by calling the function with the tail as argument). As far as the evenly divisible scenario is concerned, we only care about the zero remainder. We, therefore, use a 'don't care' since all the other possible remainders go from *0 to n-1*, where n is the integer used for division.

In addition, similar to the second presented pattern, the final output will be a list and therefore, as long as there is an element in the list, we will append the head to the recursive function step.

```
def even([]) do [] end
def even([h | t]) do
  case(rem(h,2)) do
    0 -> [h] ++ even(t)
    1 -> even(t)
  end
end
```

```elixir
def list_div(_, 0) do :error_division_by_zero end #put this code line first
def list_div([], _) do [] end
def list_div([h | t],1) do [h | t] end
def list_div([h | t], n) do
  case(rem(h,n)) do
    0 -> [h | list_div(t, n)]
    _ -> list_div(t,n)
  end
end
```

Note: One mistake that I did was that I forgot that the appending ($++$) is done between two lists and I, therefore had to write the head, as a list of one element only in order to do the operation correctly. The *odd* function has not been included since it is similar to the *even* function. The only things that have to do is swap the case operations depending on the remainder and naturally, the function's name in the recursive step.