Enkla funktioner

Azer Hojlas

January 21, 2022

Introduction

Due to a misunderstanding of the assignment instructions, I did not realize that one would not need to complete the entire assignment. I did roughly half of all functions before I could remedy this mistake, that is, to stop programming and start writing this report. Initially it was very difficult to implement the functions, but for each successful implementation the next one would be more manageable. Granted, I cannot expect that all my functions execute successfully for each type of input. Some base-cases might have been omitted and some types of input might not work, e.g negative bases for the exp-function. In general the functions should perform satisfactory for most inputs. In this report I will primarily focus on the simple list functions, as that is how far I got before the due date of the assignment forced me to start on the report. More specifically, I will go through the slightly more advanced functions that took slightly longer to implement.

Simple functions on lists

When I first started with the assignment, I went about coding in the same manner as I would for a high level object oriented language like Java. It contained plenty of if-statements, library functions and each function spanned across several lines. After being corrected by Johan I managed to make the code more compact with the use of pattern matching and function arity, thus making the code adhere more to functional programming conventions. In total, each and every function was remade twice until they satisfied said conventions. I have only saved code from the last two iterations, as I did not think that it would be useful to save the first iteration at the time.

Code

I have decided to showcase len and add respectively. Len calculates and returns the length of a list (number of elements in it) and takes said list as an input argument. Add, as the name implies, adds an element x to a list if it is not already in the list, i.e the list will not be modified if it already contains x.

```
def add(x, list) do add(x, list, list) end
def add(x, l, full) do
  case l do
    [] -> [x | full]
    [^x| _] -> full
    [_| tail ]  -> add(x, tail, full)
  end
end

def add_improved(x, l) do add_improved(x, l, l) end

def add_improved(x, [], full) do [x | full] end
def add_improved(x, [x | _], full) do full end
def add_improved(x, [_ | t], full) do add_improved(x, t, full) end

def len(list) do len(list, 0) end

def len(list, length) do
  case list do
    [] -> length
    [_| tail ] -> len(tail, length + 1)
  end
end

def len_improved(l) do len_improved(l, 0) end
def len_improved([], n) do n end
def len_improved([_ | t], n) do len_improved(t, n + 1) end
```

Further explanation of code

As you can see, each base implementation of both functions has a vague resemblance to non-functional programming, with a single function taking care of both the base cases and recursions(originally, even these functions had only one defined function and were not split in two). The functions with the "improved" suffix however, use different functions with different pattern matching and arity in order to achieve recursion and base case exceptions. Johan assisted me in implementing the first add function, so I will not explain it in detail as I do not understand the pin operator completely. Thus, for the sake of shortness, I will go through the len functions.

The first len function starts by receiving a list of which it is to compute its length. The list gets passed along to the same function but with an arity of two, along with a variable initially set to zero. This variable will be incremented for each successive element that the functions processes. The variable will be returned once the list has been recursively iterated through (hopefully set to the correct number of elements). The first base case is if the list is empty. This occurs if the list initially is empty, in which case the returned length will be zero, or when the function has iterated throughout the entire list and there are no more elements to count. If the base case isn't fulfilled, there exists a non-empty tail (lists in elixir are in essence linked lists divided into heads and tails, I will not elaborate on this any further as I expect the reader to know about them) that can be passed along to the same function without the head, with the length variable incremented. This ensures that each successive head will be counted and increment the length variable. This iteration will continue until the tail becomes an empty list, in which case the length of the list will be returned.

The improved len function also achieves what the first one does. It begins exactly the same as the first function, but diverges when it comes to the conditional execution. It does however not use a case statement. Instead, it uses pattern matching to identify an empty list. If this statement is not fulfilled, the program will jump to the third function call which in practice acts as an else statement, in which the recursion occurs in the same way as in the first function. Thus we are left with two functions, constructed in two different ways but with the same functionality. One bulky over several lines, reminiscent of non-functional programming, the other compact, distributed over several single-row functions in adherence to functional programming conventions.

Conclusion

I suspect that the main goal of this assignment was not to learn functional programming in-depth, but to ensure that we students break the object oriented habits gained during the past three years of OOP-coding. Additionally, the functional way taught me, in a very practical way, the importance of pattern matching and using the same function calls but with different arity. Through this assignment I have learned lots of useful information, and I will probably complete all the functions from the ”getting started” PDF, as they are very instructive (when I fail, that is) and are bound to give me an edge when implementing functions later on.