Lab 6 - Higher order functions

Weichen Chai

Spring 2023

Introduction

This paper seeks to answer the questions presented in”Higher order functions” for ID1019. In this assignment, we will explore how the functional language Elixir can be utilized to implement higher order functions by exploring multiple different uses of recursion and functions.

Tasks

in order to begin, we will first attempt to tackle recursive transformation of a list, where we first solve the problems presented to us in the assignment. The entire code are not given but some snippets which are the most important in displaying what I have written and proving the point I am trying to make.

Recursively transforming a list

In this task, we are first asked to implement a list where integers are doubled, this can be done by simply using recursion and multiplying each head of the list by 2 every time the function is run. After filling in the skeleton code, it is shown below:

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

In the function we used h to represent the head or the first element in the list, and the letter t as the rest of the list.

Then for the case where we add 5 to each integer included, we have a similar recursive function where the first element in the list is added by a 5 we have:

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

And at last, a program which replaces :dog with :fido on the list of elements inputted when executed, where there are two cases, one where the first element is :dog and when the first element in the list is not :dog. The implementation can be seen below:

```
if h == :dog do
    [:fido | animal(t)]
else
    [h| animal(t)]
```

It is immediately apparent that all three functions have the pattern, and we need to implement a recursive function for them to work.

At last, we try to combine them and build a function which depending on the second argument, which takes in both integers and variables.

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

Functions as data and argument

In this section we will learn about how functions can be utilized as data seen by using the examples given to us in the assignment, then we also experiment with the function as an argument where we look into a function taking in both a list and a function as arguments, we first finish the code given in this section:

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

Testing the function by when we have functions similar to the ones shown in the task, we receive the expected results, indicating that it indeed works.

Reducing a list

Under this section we will try to sum a list first by filling in the implementation:

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

This implementation is like many that we have performed above which utilizes simple recursion, then another second task is to follow the idea from apply to all which we have performed before and make a function that returns value obtained by applying function to some element recursively.

```
def fold_right([h | t], base, func) do
    func.(h, fold_right(t, base, func))
  end
```

This fold right function operates in a right to left order, some examples were chose and I have tested that this function indeed works, for a fold left order we have to consider tail recursive function and the use of an accumulating parameter.

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

Filter out the good ones

We experiment with filtering inputted lists and return only odd elements

```
  def odd([h | t]) do
    //use remainder
    if rem(h, 2) == 1 do
    //move to next if odd
      [h | odd(t)]
    else
    //disregard and simply test odd of next
      odd(t)
    end
```

At last we try to implement a similar function named filter that takes in a list and a function with returning true or false.

```
def filter([h | t], func) do
   if func.(h) ==  true do
       [h | filter(t, func)]
     else
        filter(t, func)
     end
 end
```

The function above works with any filtering function, such as odd, even, or even greater than and less than.