Higher Order Functions

Ayda Ghalkhanbaz

Spring Term 2024

Introduction

This assignment builds upon the previous one; where we had 3 groups of operations on lists. This report covers the implementation of higher order functions and ends with results of run tests on the program.

Method

Higher order functions are functions that accept other functions as arguments and operate on them. This assignment is about implementing such functions that take our previous list operators as arguments and do the repetitive parts. In this way, we avoid repeating the same algorithm for each function, just as we did in the previous assignment.

So the purpose is to categorise these functions and implement a higher order function to achieve the algorithm. As before, we have three different groups which we will delve into the implementation details of each in the upcoming sub sections.

Map

The underlying logic of this function is quite straightforward; we want to traverse the entire list and do some operations such as multiplication, addition or subtraction on each element of the list and then return the list with its updated elements. The `map/2` function takes a list and a function as arguments, operates on each element of the list using its function argument and recursively calls itself to traverse the list.

The code snippet below illustrates `map/2` function.

```
def map([], _) do [] end
def map([head|tail], function) do
  [function.(head) | map(tail, function)]
end
```

Reduce

The second function somehow calculates an attribute of the list using its elements and finally returns a single value. This attribute can be the sum of all elements in the list or the length of the list. There's of course several ways to implement this function but in this assignment we chose to do it in two different ways, i.e. the tail recursive and the body recursive implementation. The difference between them is more obvious when dealing with large amounts of data, then the tail recursive implementation is less effective since it has to go through the list all the way down and back.

For more clarification, the code snippet below demonstrates the implementation details of this function in both approaches.

```
# tail recursive
def reducel([], acc, _) do acc end
def reducel([head|tail], acc, function) do
  reducel(tail, function.(head, acc), function)
end
# body recursive
def reducer([], acc, _) do acc end
def reducer([head|tail], acc, function) do
  function.(head, reducer(tail, acc, function))
end
```

Filter

This function takes a list and a function and applies a filter on the list. For instance, based on the operation it may return a new list with all odd or even elements. Also, this function has been implemented in three different ways, where two first implementations keep the original order of the elements while the third one is the regular recursion and reverse the order of the elements.

Below you'll find the code for the right and left append implementation for the `filter/2` function.

```
# right append (filter) (keeps the order)
def filterr([], _) do [] end
def filterr(list, function) do
  filterr(list, function, [])
end

def filterr([], _, acc) do acc end
def filterr([head|tail], function, acc) do
```

```
    case function.(head) do
    # ++ concates two lists
      true -> filterr(tail, function, acc++[head])
      _ -> filterr(tail, function, acc)
    end
  end

  # left filter (keeps the order)
  def filterl([], _) do [] end
  def filterl([head|tail], function) do
    case function.(head) do
      true -> [head|filterl(tail, function)]
      _ -> filterl(tail, function)
    end
  end
```

As the last step of this assignment, we have re-implemented the operator functions from the previous task, so that they can be applicable on the higher order functions. Below you'll find three different operators that utilise each of these higher order functions.

```
# incrementing elements by a number
  def inc(list, number) do
    func = fn(a) -> a+number end
    Reduce2.map(list, func)
  end

# production of all of the elements
  def prod(list) do
    func = fn(a, b) -> a*b end
    Reduce2.reducel(list, 1, func)
  end

# a list with all divisable elements by number
  def divz(list, number) do
    func = fn(a) -> rem(a, number) == 0 end
    Reduce2.filterl(list, func)
  end
```

In addition to the previously implemented operators, another function is implemented where it returns the sum of the squared elements less than the given number. This function employs the `map/2` function to apply the operation on elements in the list and then calls `sum/1` to do the addition operation on the elements. The following code snippet shows the implementation of this function.

```
  def less_square(list, number) do
    func = fn(x) -> (if x < number do x*x else 0 end) end
    sum(Reduce2.map(list, func))
  end
```

Result

In this section, we will see a run test of some of the functions, where the `testPipe/3` function utilises pipe operators.

```
def testPipe(list, x, y) do
    list |>
      divz(x) |> # find all elements divisable with x
        inc(y) |> # increment the elements with y
          lengthz() # return the length of the final list
  end
```

................................Run Test ....

```
iex(9)> OP.testPipe([1,2,3,4,5,6,7,8,9,10],2,3)
5
```