Functional Programming Assignment Report

Bohdan Tymofieienko B.S.
b.tymofieienko@student.fontys.nl
4132645

Mikolaj Hilgert M.H.
m.hilgert@student.fontys.nl
4158385

March 11, 2022

Contents

1 Merge Sort Week 4

In the first assignment for this week we were tasked to implement a merge sort algorithm in Elm. To do so we obviously did some research. We decided to split our problem into sub problems and solve them one by one. We started with "split" function since it is one of the fundamental operations in merge sort.

We used list recursion only to split the list into two parts. There are multiple ways to do so, however we decided to take every second element of the list and in this way divide the list. In essence we slice two elements from the list and add each of them to different lists. If it happens that there is only one element left then we simply add it to the left list.

```
split : List a -> List a -> List a -> ( List a, List a )
split list left right =
  case list of
    [] ->
      (left, right)
    x::[] ->
        split [] (x :: left) right
    x :: y :: xs ->
        split xs (x :: left) (y::right)
```

Going further we needed a function that merges two lists with regard to value of an element of the list. By this we mean that if there exists two lists with elements x_i y_i, than we append the smallest of those to resulted list. As following,

```
merge: List Int -> List Int -> List Int
merge left right =
    case (left,right) of
        ([],[]) -> []
        (_,[]) -> left
        ([],_) -> right
        (x::xs,y::ys) ->
            if x < y then
                x :: merge xs right
            else
                y :: merge left ys
```

Sorting algorithm looks fairly simple, since with the use of recursion we divide the list into smaller slices until they reach the base case (in this case list with one element). And after that merging procedure comes to places and we merge list with regard to which element is larger. This is an example of "Divide and conquer" algorithm strategy.

```
msort: List Int -> List Int
msort input =
    case input of
        [] -> input
        [_] -> input
        _ ->
              merge (msort (first (split input [] [] )))
                    (msort (second (split input [] [])))
```

2 Modelling Math functions (part 1)

The second task was to implement a function that would graph mathematical equations using Elm.

The first thing that we had to define was a custom Type. This type is of function, and it is used to specify the variants of functions we may have.

```
type Function
    = Poly Function Float
    | Mult Function Function
    | Div Function Function
    | Plus Function Function
    | Minus Function Function
    | Const Float
    | X
```

Next, as per the assignment we must implement a **print** function, that takes as an input an equation in the form using the aforementioned Function type. And returns a human readable form. For example (print (Mult (Const 2) X)) would return in (2*x). We tried to replicate the syntax from the Reader. With recursion, it made it possible to pass a function that is composed of other functions, as with case matching, we may recursively call the implementation of given type.

```
print: Function -> String
print func =
    case func of
        Poly base power -> "("++(print base)++"^"++fromFloat power++")"
        Mult left right -> "("++(print left) ++ "*" ++ (print right)++")"
        Div top bottom -> "("++(print top)++")" ++ "/" ++ "("++(print bottom)++")"
        Plus left right -> "("++(print left) ++ "+" ++ (print right)++")"
        Minus left right-> "("++(print left) ++ "-" ++ (print right)++")"
        Const value -> fromFloat value
        X -> "x"
```

Next, we implemented the **eval** function, This was done in a similar recursive manner using case matching, but now rather than creating a string we do a mathematical calculation.

```
eval: Float -> Function -> Float
eval x func =
    case func of
        Poly base power -> (eval x base) ^ power
        Mult left right -> (eval x left) * (eval x right)
        Div top bottom ->  (eval x top) / (eval x bottom)
        Plus left right -> (eval x left) + (eval x right)
        Minus left right-> (eval x left) - (eval x right)
        Const value -> value
        X -> x
```

Finally, we may implement the graphing function. This function relies on the aforementioned eval function to find the co-responding y value to a given x. This graph function takes as parameters the given function itself, and the x and y bounds. This function works by working through the given Domain. Meaning from the bottom of the given x bound to top. Whilst we are still within the given domain, then we use the helper function drawline to draw the graph for the given x, by

evaluating the y value of the function for the given x. Then recursively calling for the next line (see xmin+1).

```
graph: Function -> Float -> Float -> Float -> Float -> String
graph func xmin xmax ymin ymax =
    if xmax == xmin then
        ""
    else
        (drawLine (eval xmin func) ymax ymin)++ "\n" ++
        graph func (xmin + 1) xmax ymin ymax
```

The drawLine function works on the premise of recursively going through the given range, and choosing the character based on whether it is smaller or larger than the y value given by the eval function. Meaning, if we start at the bottom bound of the Range, we check whether that value is smaller than the eval (meaning the y point at which the function is). The character chosen is based on whether the given value is larger or smaller than the eval.

```
drawLine: Float -> Float -> Float -> String
drawLine value upper current =
    if current < value && current <= upper then
        "*" ++ (drawLine value upper (current+1))
    else if current <= upper then
        "-" ++ (drawLine value upper (current+1))
    else
        ""
```