Programming Assignment 2

CS-420 Spring 2024

Juan Pablo Zendejas

March 15th, 2024

For this programming assignment, I was tasked with creating a variety of functions for different purposes to practice functional programming concepts like fold/reduce, recursion, and function composition. In all, I managed to get a full score on the autograder and I felt that I had created acceptable solutions that are clean and easy to understand.

While I didn't work with anyone specifically on this project, I did end up searching for help online when I felt I had hit a wall. For example, I wanted to use only one `foldr` call on my `sepConcat` function and I was having trouble creating a lambda function that would work. I looked at some StackOverflow[1] answers for some guidance and I found I was close, but I just had to re-arrange my use of the `++` operator to follow the ordering of the `foldr`. I also found out about the trick of using `zip [0..] xs` to get the index of each element of a list, which was useful for the final problem.

Now, we will look at each function and my solution.

1 Part A: Basic Haskell Prelude

These functions are simple implementations using basic Haskell prelude functions.

1.1 myMod

The idea is fairly simple. Use `div` to perform integer division of x and y. Then, we can multiply it by y and subtract that from x to get the remainder of the division, which is the modulus operator.

```haskell
myMod :: Int -> Int -> Int
myMod x y = x - (y * div x y)
```

[1] https://stackoverflow.com/a/44101237

1.2 toDigit

This function uses recursion. If the input is □0, then we just return an empty list. The recursive step then splits the integer into the first digit using `myMod`, and then the rest of the number using `div`. Finally, the digit is appended to the end and toDigit is called recursively to prepend its result.

```
toDigit :: Int -> [Int]
toDigit n | n <= 0    = []
          | otherwise = toDigit (div n 10) ++ [myMod n 10]
```

1.3 reverseList

Another recursive function. To reverse the list, I use a pattern matching to split the first element of the list out. Then, just prepend the reverse of the rest of the list to the first element.

```
reverseList :: [a] -> [a]
reverseList [] = []
reverseList (x:xs) = reverseList xs ++ [x]
```

1.4 sumList

Follows a similar strategy. Pattern-matching to split the first element off, then add it to the sum of the rest of the list. This recursive call is ended by a base case of 0 for an empty list.

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = sumList xs + x
```

1.5 toDigitRev

Just a function composition.

```
toDigitRev :: Int -> [Int]
toDigitRev n = reverseList (toDigit n)
```

2 Part B: Folding Functions

These functions use the `foldr` prelude function to perform operations on a list.

2.1 myDouble

The idea was to use `foldr` with a list of the parameter twice, so foldr could add them together.

```
myDouble :: Int -> Int
myDouble n = foldr (+) 0 [n,n]
```

2.2 doubleEveryOther

This function did not have to use `foldr`. Here, the idea was to use recursion. The recursive case would grab the first two elements and the rest of the list. Then, the 2nd element would be doubled, and the elements would be appended in order but with the doubleEveryOther recursive call for the rest of the list.

```
doubleEveryOther :: [Int] -> [Int]
doubleEveryOther [] = []
doubleEveryOther [x] = [x]
doubleEveryOther (x:y:xs) = x : myDouble y : doubleEveryOther xs
```

2.3 mySquare

Similar to myDouble, but using multiply instead of adding them.

```
mySquare :: Int -> Int
mySquare n = foldr (*) 1 [n,n]
```

2.4 sqSum

Here, we use a custom lambda function. In addition, because the last parameter is the list, we can also use partial function application or currying to omit the name of the first argument and make the declaration cleaner. The lambda function just adds the square of x to the accumulator of `foldr`.

```
sqSum :: [Int] -> Int
sqSum = foldr (\x acc -> mySquare x + acc) 0
```

2.5 sumDigits

This function needs to add the digits of all the numbers in the list. Here, the custom lambda function uses the previous `sumList` and `toDigit` functions. It adds the sum of the digits of each element to the accumulator.

```
sumDigits :: [Int] -> Int
sumDigits = foldr (\x acc -> acc + (sumList $ toDigit x)) 0
```

2.6 sepConcat

Concatenate a list of strings while interspersing a separator string in between each string. For this, the custom lambda function will prepend the string to the accumulator, but only put the separator in between if there has already been an element added to the accumulator. This way, we avoid the off-by-one error of appending the separator to the end of the final string.

```
sepConcat :: String -> [String] -> String
sepConcat sep = foldr (\x acc -> x ++ if acc == [] then acc else sep ++ acc)
 ↪  ""
```

3 Part C: Credit Card Problem

This is an application of some of the functions I've written to create a function that validates a credit card number using the Luhn algorithm. Double every other digit of the card number starting from the end (reversed). Take the sum of those digits. If the sum is equal to 0 mod 10, it is a valid credit card number.

```
validate :: Int -> Bool
validate n = myMod (sumDigits (doubleEveryOther (toDigitRev n))) 10 == 0
```

4 Part D: Sorting Algorithms

The task for this section was to implement some sorting algorithms in a functional manner.

4.1 splitHalf

For the `splitHalf` function, it needs to split a list into a tuple where the left and right tuple each have half of the list. To accomplish this, I first created a generic `mySplit` function that splits onto an index. This is accomplished by taking in an index and a pair of lists. The index is the number of elements from the right pair that should be moved into the left pair. So, every recursive call will move the first element from the right list into the end of the left list. Then, mySplit will be called again with `n` decremented. The base case, when `n = 0`, is to just return the pair.

Finally, `splitHalf` just calls the `mySplit` function with `n` equal to half the length of the list.

```
mySplit 0 p = p
mySplit n (xs,(y:ys)) = mySplit (n-1) (xs++[y], ys)

splitHalf :: [a] -> ([a], [a])
```

```
splitHalf xs = mySplit (div l 2) ([], xs)
            where l = length xs
```

4.2 mergeList

This is where it starts to get complicated. `mergeList` is a function that receives two lists of pairs that are sorted by the 2nd element of the pair, and merges them into one big sorted list. The way I thought about this problem after talking with the professor is to consider the first element of each list. Then, I just let the smaller one go first and recurse. To accomplish this, I used a bunch of pattern matching and a guarded statement. Here, `axs` and `ays` are the whole lists. Then, `px` and `py` are the pairs that I should prepend. `xs` and `ys` are the rest of the list without the first element. Finally, `x` and `y` are the element the list is sorted by. The guard statement just compares x and y, and uses that information to prepend the pair with the smaller sorting value.

```
mergeList :: Ord b => [(a, b)] -> [(a, b)] -> [(a, b)]
mergeList [] pys = pys
mergeList pxs [] = pxs

mergeList axs@( px@(_,x) : xs) ays@( py@(_,y) : ys)
    | x <= y    = px : mergeList xs ays
    | otherwise  = py : mergeList axs ys
```

4.3 mergeSort

Finally, I got to implement the merge sort algorithm. I first created some helper functions. `applyPair` applies a function f with the parameters given by a tuple. `applyEachPair` applies a function f to both elements of a tuple, and returns a new tuple. Then, `mergeSort` puts it all together and applies merge sort to each half of the list; then applies mergeList to the pair of the sorted halves. The base cases end the recursion of mergeSort, since a list of 0 or 1 elements is sorted.

```
applyPair :: (a->b->c) -> (a,b) -> c
applyPair f (x,y) = f x y

applyEachPair :: (a->b) -> (a,a) -> (b,b)
applyEachPair f (x, y) = (f x, f y)

mergeSort :: Ord b => [(a,b)] -> [(a,b)]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = applyPair mergeList (applyEachPair mergeSort (splitHalf xs))
```

5 Part E: BigInt

Here, I was tasked to work with a new type BigInt which is a list of integers.

5.1 clone

Clone takes a value and a number, and creates a list with that value repeated n times. So, a simple recursive solution is to use the cons operator and call clone again with n decremented. The base case is when n = 0, which returns and empty list.

```
clone :: a -> Int -> [a]
clone _ 0 = []
clone x n = x : clone x (n-1)
```

5.2 padZero

This function will take two `BigInts` and return a pair of new `BigInts` that have the same length. Here, I just made a quick helper function `clonez` that clones zero n times. Then, I use a guarded expression to prepend zeros based on the difference of the lengths. If `ys` is shorter, it gets `xl - yl` zeros prepended, and vice-versa for `xs`.

```
clonez = clone 0

padZero :: BigInt -> BigInt -> (BigInt, BigInt)
padZero xs ys | xl > yl    = (xs, clonez (xl-yl) ++ ys)
              | xl < yl    = (clonez (yl-xl) ++ xs, ys)
              | otherwise  = (xs,ys)
            where xl = length xs
                  yl = length ys
```

5.3 removeZero

Kind of like the opposite of `padZero`. Takes a `BigInt` and removes leading zeros that have no value. This is just a simple recursive pattern matching. If the first element is zero, remove it and call `removeZero` again. If we can't pattern match a zero, just return the list.

```
removeZero :: BigInt -> BigInt
removeZero (0:xs) = removeZero xs
removeZero xs = xs
```

5.4 bigAdd

This function will add two `BigInt`s. To implement this, I delegated to a helper function `bigAdd'`. The reasoning was so I could use tail recursion and bring the carry value over each time.

First, `bigAdd'` will take two reversed `BigInt`s, along with a carry bit. We take the sum of the first integers from the lists, along with the carry sum. Then, we take the first digit of the sum and prepend it, and then recursively call `bigAdd'` with the carry bit being the sum divided by 10, and the rest of the two lists. The base case, then, simply takes empty lists and returns the carry bit.

The actual implementation of `bigAdd` pads the incoming `BigInt`s with zeros, reverses them, and calls `bigAdd'` with a carry of 0. Finally, at the end, the result is reversed and trimmed of zeros.

```
bigAdd' [] [] n = [n]
bigAdd' (x:xs) (y:ys) n = (myMod sum 10) : bigAdd' xs ys (div sum 10)
                        where sum = x+y+n

bigAdd :: BigInt -> BigInt -> BigInt
bigAdd xs ys = removeZero $ reverseList $ bigAdd' (reverseList zxs)
 ↪  (reverseList zys) 0
               where (zxs,zys) = padZero xs ys
```

5.5 mulByDigit

I followed a similar strategy to `bigAdd`. Essentially, the helper function `digMul'` will take a reversed list, and then multiply the first digit by some factor q. Then, the carry bit is added, and the sum is split into the first digit and the other digits. The first digit is prepended to the recursive call of `digMul'` with the rest of the list and the same factor.

```
digMul' [] _ n = reverseList $ toDigit n
digMul' (x:xs) q n = (myMod sum 10) : digMul' xs q (div sum 10)
                        where sum = (x*q)+n

mulByDigit :: Int -> BigInt -> BigInt
mulByDigit q xs = removeZero $ reverseList $ digMul' (reverseList xs) q 0
```

5.6 bigMul

Finally, the `bigMul` function will take two `BigInt`s and multiply them together. This function follows a basic long multiplication algorithm. That is, I take each digit from the second number and multiply it digit-by-digit to the first number. Then, based on its position, I add zeros to the end of the result. To implement this in Haskell, I used a helper function `megaSum` that recursively adds a list of `BigInt`s using the previously

created `bigAdd` function. To create the list of numbers to add, I had to reverse the 2nd list `ys` into `rys`. Then, by using the `zip` method, I combined each element of `rys` with its index. Since the list was reversed, this index is the number of zeros to add onto the sum. Each element of the list passed to `megaSum` is that digit of `ys`, `q`, passed to `mulByDigit` with the whole `BigInt xs`. Finally, zeros are appended to the end based on the index. The sum of all of these products is the final result.

```
megaSum :: [BigInt] -> BigInt
megaSum [] = [0]
megaSum (x:xs) = bigAdd x $ megaSum xs

bigMul :: BigInt -> BigInt -> BigInt
bigMul xs ys = megaSum [mulByDigit q xs ++ clone 0 i | (i,q) <- zip [0..]
 ↪  rys]
               where rys = reverseList ys
```

6 Results

The Autograder on EDORAS gave me full marks.

Results

```
------------------------------------------------------------------------
    Results: [90.00/90.00] | [100.00%]
------------------------------------------------------------------------
```