Assignment 2

Klement/cgx702

1 Reflections on group work and sources

1.a

My group members are Lucas Marshall, Noah Skovborg and Oliver Nygaard

1.b

We meet up Tuesday and talked about the assignment and the concepts we though we could use for the assignment, we focused particularly on Pascals triangle, as we deemed this the hardest part. This talk was heavily focused on getting a common understanding of the assignment, not so much actual common code. Afterwards, because of wishes from group members, most of the communication was handled through messenger, which led to the group feeling less connected, than it might have if we had meet up together throughout the week. This was this sufficient to solve the assignment, as we had all understood the assignment, before we went of and sat with it alone.

1.c

I used Fsharp.Core and slides, practices and worksheet from week 4, to look at how and which higher order functions i could use to create the array i use in 2.B.

I used work i had done in the deliberate practice and the worksheet from week 7, as a jumping of point for the higher order functions i defined for MyList in Question 3.

2 Pascal's triangle

2.a

I need to create a function which using recursion given a n and k, which are whole numbers bigger than or equal to 0, as an input pair, and calculates how many unique combinations can be made when choosing k of n. I will call this function pascal. As specified in the question description, the input is a pair of integer (n, k) and the output is an integer.

This function used pattern matching with (n, k). The first pattern checks if these are even valid inputs. This is because (n, k) are only valid input if they are inside the triangle, meaning no negative integers are not in the triangle. We can also choose n of k, if k is larger than n, this input is also out of the triangle. The first pattern checks if any of these are true and if they are then we return 0. We then check if we are on the edges meaning of $k = 0$ or if $n = k$, these are two separate patterns, but they both return 1. Now we get to the recursive case, here we return of the sum of the two elements above the element we are checking. In code we call the pascal function with $(n - 1, k - 1)$, which is the element up and to the left, and with $(n - 1, k)$, which is the element up and to the right.

2.b

This function is very similar to the function from a, but now we need to fill out pascals triangle, which is a int array array, and return the value of the (n, k), which will be a position in the triangle. This function takes a pair of integers (n, k) as input and the output is an integer. I choose to call this function pascalNoRec.

This function starts with checking if these are valid inputs, $n \geq k, n \geq 0, k \geq 0$ and returns 0 if they aren't valid inputs. Next i generate a triangle using:

```
let mutable triangle = Array.init (n+1) (fun index -> Array.
    zeroCreate (index+1))
```

The Array.init creates an array of elements (n+1) and applies a function to each of these rows, here we create another array, filled with zeroes which has the current index + 1 elements. We need to write n + 1 and index + 1, because pascal's triangle begins at zero for both row and column. This generates a triangle which is filled with zeroes with n number of rows. Pascal's triangle begins at 0, so the first row is row 0. The rest of this function then fills out the elements in the triangle according to the two properties.
I do this using two for loops. The first for loop keeps track of the rows, from 0 to n and the other keeps track of the columns from 0 to row. Inside the columns for loop, is where all the action happens. Here we check if we are on either edge of the triangle meaning, columns = rows or columns = 0, if we are on an edge we assign our current position the value 1. If we aren't on the edge of the triangle, we set our current position to the sum of the values at the position (row-1, column-1) and (row-1, column). We slowly from the top down fill out the value of the binomial coefficients from the top of the pyramid all the way down, until we reach the end of the n'th row. This means we might have filled out some element which we didn't need to calculate the given (n, k). // After we have reach the n'te row we then return the value of the (n, k) position.

2.c

My functions already accounted for whole numbers outside of the triangle, these return 0. It says on the Wikipedia page in footnote 4, that the binomial coefficient conventionally is set to 0 if the values are outside the triangle. My function doesn't handle inputs which are not integers, these values simply give an error and the functions don't run. This could be improved if we used Options or failwith, but i didn't have the time to make it work.

2.d

I have designed 7 functions the 4 that are specified in the question description and 3 additional functions. 6 of these functions all have the same type signatures:

```
int * int -> bool
```

These functions test if the functions have Properties 1 or 2 and that they return 0 for an input outside of the triangle. These return false if the input isn't testing for that specific test cases specification or if the function returns the wrong thing. Lets take one of these function and look closer at since they all pretty much look exactly the same.
The function hasPropertyOne function takes a pair as inputs and checks if $k = 0$ or $k = n$ if they are we run the function and return the true if the output is equal 1 and false if not. The function also returns false if $k \neq 0$ or $k \neq n$.
The only thing that changes from the function hasPropertyOne, hasPropertyTwo and the function inputOutOfTriangle is the conditions we check. For hasPropertyTwo we check if $n > k$ or $k > 0$ and if the $(n, k) = (n-1, k-1) + (n-1, k)$. And for inputOutOfTriangle we check if $n < 0$ or $k < 0$ or $k > n$ and if the output is equal 0.
I also have a function test, which simply is for my own pleasure and returns a string "PASSED" or "FAILED", depending on the result from the testing functions, which we then print in the terminal along with the pair we tested for.
We run through a list of pairs for each function that we test and print the result of. The list of inputs is unique and test the specific Property we are testing.

I have choosen values to test each Property:

```
let propertyOneTestValues = [ (0,0); (6,0); (6,6) ]
let propertyTwoTestValues = [ (5,2); (6,3); (2,1) ]
let invalidTestValues = [ (-1, 0); (3, -1); (3,5)]
```

My inputs tests each of the different cases for each of the different properties we are testing. These tests make sure that the functions return values so that the specifications fit. The inputs are partitioned, so that we are only checking for one thing at a time, we don't have a function that overlaps and checks for multiple cases. I should probably also have tests for floats, but since i know this will give a runtime.error, since my functions don't handle this case i don't test for it. This is again because i didn't have the time to make it work with options and proper error handling.

2.e

I can be pretty sure that my function works for all possible values of n and k that are specified in the specification. Since the specification says, that the numbers need to be whole numbers equal to or greater than 0.

3 Implementing helper functions for lists

3.a

The specifications are in myList.fsi

3.b

I needed to define a function, called take, that takes and integer n and a MyList and returns a MyList of the first n elements in the original MyList. I use pattern matching on both the inputList and n. First i check whether n is less than or equal to zero and return an Empty list if it is. I then check whether the MyListist is empty and return an empty MyList if it is. If either of those base cases aren't true, then we concatenate the function call with n-1 and tail as parameters onto the head.

I need to define a function, called drop, that takes and integer n and a MyList and returns a MyList without the first n elements of the original MyList. I use pattern matching again on both the inputList and n. First i check whether n is less than or equal to zero and return an Empty list if its is. I then also check whether the MyList is empty and return an empty MyList if it is. If either of these base cases aren't true, then we call the function again with n-1 and tail as the parameters.

I need to define a function, called length, that takes a MyList and returns the number of elements in the list. I use pattern matching on the inputList, First i check whether the MyList is empty and if it is i return 0. Then if the list is not empty i plus 1 to the return value of the function call again with only the tail.

I need to define a function, called map, that takes a MyList and a function and returns a MyList with the function applied to each element. Again i use pattern matching on the inputList. I check if it is empty and if it is i return an empty MyList. If the inputList is not empty i apply a function onto the head of the MyList and concatenate the return value of the map function called again, with the same function and the tail as paramters.

3.c

Examples are in MyList.Examples.fsx

3.d

the map function doesn't work if it is given a function with the type signature appliedFunc : string -> string and it is a int MyList. The input of appliedFunc needs to match with the type of elements in MyList. This is also specified in the specification.