Assignment 1

Klement/cgx702

1 Bank accounts

1.A

I represent a customer using a tuple of two strings. The first string represents the name of the customer and the second is a customerID. The Account is represented with a string, which is the accountID, this is used to both identify the account owner, the specific account and whether the accunt is a checking or saving.

```
type konto = (string * int)
type kunde = (string * string)
```

The first char of the accountID signifies the account owner, the second char indicates, the account number of that specific customer and the last char is either C or S, meaning either Checking or Saving. Example:

```
("Dupont", 5)
("51C", 300.00)
("51S", 780.50)
("52S", 1200.00)
```

The customer Dupont, has three accounts, 2 savings and 1 checking. These values are stored in two different lists, a list of all customers and a list of all accounts, this means that i don't have to store the specific account with the specific customer, as this is implicit in the account name.

1.B

The function needs to able to transfer funds from a saving to checking of the same user. The function needs to check whether, the saving account has enough funds to transfer and whether both the accounts belong to the same customer.

We call this function internalTransfer, since this function only handles internal transfers.

As an example, we could run this function transferring from one of Dupont's saving accounts to his checking account.

```
internalTransfer "51C" "51S" 1000.0
```

If we call the function like this we would expect to get the output:

```
Overdraw is no allowed
Din saldo er 780.5
```

This is because the balance on Dupont's saving account is only 780.5 If he instead tries to transfer, the same amount, from his second saving account which has 1200.00 we would get the input:

```
internalTransfer "51C" "52S" 1000.0
```

and the output:

```
Senders balance before: 1200.0
Senders balance after: 200.0
Recipients balance before: 300.0
Recipients balance after: 1300.0
```

The function takes three inputs, a string that is the accountID of the recipient, a string that is the accountID of the sender and a float that is the amount we are transferring. The function doesn't really have an output, as we print everything inside the function, so the output is unit.

First we need to check whether the savings account has enough money, i have, using ken's method, defined a helper function which, given a accountID, finds the balance of the account, if no account exists, it should simply return 0. I call this function findBalance.

Lets as an example look at what we would expect the output to be if we give it two different accountID's, as input. Lets take Tintin's first checking account "11C" and another account "89C", which does not currently exist.

```
findBalance 11C
Return value: 1451.67
findBalance 89C
Return value: 0.0
```

The function takes, the accountID, a string as input and outputs the balance as a float. The function looks through the entire list of accounts for an account with a matching ID. It then returns the balance part of the account Details. If no account is found, we set this balance to 0.0.

```
let findBalance (accountID : string) : float =
    match alleKonto |> List.filter (fun (accID, _)
        -> accID = accountID) with
    | [(_,b)] -> b
    | _ -> 0.0
```

We can these use this helper function to check if the balance is less than the amount we are trying to transfer, and if the account contains fewer funds than expected we print "Overdraw is not allowed" and also the balance of the sender account.

After that we check if the two accounts belong to the same person, i simply compare, the first char of the accountID of both accounts which i use this to signify the account owner.

If the accounts don't belong to the same person, i print "This is not an internal Transfer" else i simply print the old and new balances of both the checking and savings account.

1.C

The function needs to first extract all the saving accounts from the list of all the accounts and then we need to add the interest to all the balances of the saving accounts.

I will call this function addInterest

Lets take only Dupont's accounts as an example the input would be:

```
addInterest 0.15
```

the function takes interest we want, since we already have the allAccounts defined we don't need to pass it through. The expected output of this would be, I print the checking accounts before the interest is apllied inside the addInterest function, but the function simply returns the new saving account list, which I print afterwards:

```
All checking accounts before interest:
[("51S", 780.5); ("52S", 1200.0)]
Return value: [("51S", 897.575); ("52S", 1380.0)]
```

This function only takes one input, the interest as a float, if we wanted 15% interest, we pass 0.15 as an argument in the input. The only downside is that all saving accounts will have the same interest applied to them. The function now produces an output which is a list of Accounts. For this function, i start with finding all the saving accounts in the list of all the accounts. I use a list filter, that checks if the last char of the accountID string is equal to S.

```
let savingAccounts =
    allAccounts
    |> List.filter (fun (accID, _)
        -> accID.[accID.Length-1] =  'S')
```

I then print the list of saving accounts before interest, for fun. I then use another higher order function, List.map, making a new list with the same account ID and the balance after interest.

```
savingAccounts
    |> List.map (fun (accID, balance)
        -> (accID, balance * (1.0 + interest)))
```

The List.map function now returns the a new list, with the altered saving accounts. I then print this when i call the function in the bottom of bank.fsx. This only adds the interest once, but the function could easily be modified to do this again and again.

2 Groups

2.A

The function needs to take a list of 100 integer/student ID's and put them into pairs.

I call this function createPairs. As an example let's say its only 10 students as input that would mean the output is:

```
[(0, 1); (2, 3); (4, 5); (6, 7); (8, 9)]
```

This function takes the list of Student ID as input, which is an int list. The output is int list list. This is because we can risk the groups having a variable length, if there is an uneven number of students, the last three students will then be in a single group.

This is a recursive function where we use pattern matching. We check if the function is exactly three elements long, in the first pattern, if it is we return a list containing the first, second and third element. This is in case the number of students are uneven, we then create a group containing the last three people. We then take the first two parts of the list, create a int list, we then concatenate that onto the output of createPairs function, that is called again with the tail as input. If the input matches neither of those two cases meaning it is empty or there is just one person in the students list, simply return an empty list.

```
let rec createPairs (students : int list) =
    match students with
    | first :: second :: third :: []
        -> [[first;second;thrid]]
    | first :: second :: tail
        -> [[first;second]] @ createPairs tail
    | _ -> []
```

2.B

This is a function that needs to generate 4 different lists of the four different study lines, from a base of 100 students. We can create the pairs using the createPairs function, once we have the students sorted into their respective study line.

I will call the function that creates these study lines, createLineList.

lets take the example of doing it with only 10 student ID's. we would expect to get

```
[[0;4;8]]
[[1;5;9]]
[[2;6]]
[[3;7]]
```

The function takes as input an integer, which is the study line number and the integer list of all student ID's. The output of this is a int list, which we can then pipe into the createPairs function.

I created the function so that it makes the list based on the the studentID modulo 4. We simply make a list based on what it equals, so all the ones where. This does mean we have to call this function 4 times.

```
    let createLineList (students : int list) (lineNum : int) =
        List.filter (fun index -> index % 4 = lineNum) students;;
```

This means that it splits the 100 students into four even study lines, from which we can then generate the pairs. This leaves one without a pair, from each study line, this is a problem i couldn't figure out how to solve.

The first of my three issues, was how to represent the students, when they needed to be in four different study lines. I first thought about doing a tuple with two integers, one for student ID and one for the students study line, kind of like we did for the bank. But i thought it would be weird to generate a list of 100 predefined tuples and couldn't figure out how to automate that process. I also couldn't figure out what to do with the leftovers from each study line, i don't know how to solve this issue, my best idea is to create a pair with a single student, by checking in the create pairs function if there is only one student left in the list. I just didn't have time to do this. The last is the concept of duplicates pairs, this is solved by simply going through the lists, this does create identical lists every single time.

3 concepts and syntax

3.A

Question: What is a higher order function?

Answer: A higher order function is a function that takes a function as a parameter or returns a function.

3.B

Question: What is a type?

Answer: A type is a grouping of values, for which there are associated operations.

3.C

Question: What are the advantages of recursive functions based on your experience from the weekly activities?

Answer: Using recursive functions, means that you can instead split the problem into one or more repetitive tasks, done with small alterations between each task.

A great example, from class, is using recursion to find the factorial of a number, instead of nested if statements or manual checking, we split the task, into a small issue that can be done over and over again, which is multiplying with the previous number, this way we simply go back through the numbers doing it for all until we reach the base case, in this case 0 or 1. With factorial this is really easy, because of the definition of a factorial number is so recursive out of the box.

Recursion isn't always as linear a journey as when finding the factorial of a given number. Some times we have to use more logic or math to figure out what to call the recursive case with. An example of from class is finding the greatest common divider of two numbers. Instead of calling the function again with n - 1. We instead call the function with the divisor and the remainder of the division between the two numbers, until the remainder becomes 0, which is our base case and also means we have found the greatest common divider.

using recursion means we can express in few lines, what would otherwise have taken more complex code and logic to express in many lines.