

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 account is a checking or saving.

```
1     type konto = (string * int)
2     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:

```
1     ("Dupont", 5)
2     ("51C", 300.00)
3     ("51S", 780.50)
4     ("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 be 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.

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

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

```
1     Overdraw is not allowed
2     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:

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

and the output:

```
1     Senders balance before: 1200.0
2     Senders balance after: 200.0
3     Recipients balance before: 300.0
4     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 defined a helper function which finds the balance of an account. This function is defined as findBalance, and takes a string as input and outputs the balance as a float. The function looks through the entire list of accounts and finds the account with the matching ID. It then returns the float part of the account details, which is the second part of the tuple. If it returns nothing then the balance is set to 0.0.

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

If the balance is less than the amount we are trying to transfer we print "Overdraw not allowed" and 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 since 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 w

I will call this function addInterest

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

```
1      add interest 0.15
```

the function takes interest we want, since we already have the allAccounts defined we don't need to pass it through. The output of this would be:

```
1      All checking accounts before interest:
2      [("51S", 780.5); ("52S", 1200.0)]
3      All checking accounts after interest:
4      [("51S", 897.575); ("52S", 1380.0)]
```

This function only takes one input, the interest which is a float. The only downside is that all the saving accounts have the same interest. This doesn't produce an output, since we don't return a value, that means the output is just a unit.

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.

```
1      let savingAccounts =
2          allAccounts
3          |> List.filter (fun (accID, _) -> accID.[accID.Length-1] =
4              '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.

```
1      savingAccounts
2      |> List.map (fun (accID, balance) -> (accID, balance * (1.0
3          + interest)))
3      |> printfn "All saving accounts after interest: %A"
```

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:

```
1      [(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 a list of (int * int) tuples.

This is a recursive function where we use pattern matching. We take the first two parts of the list, create a tuple pair and then call the function with the tail. We also check if the list is empty then we simply end the function and return the list with the pairs. In practice this looks like:

```
1      let rec createPairs (students : int list) =
2          match students with
3          | first :: second :: tail -> (first, second) :: createPairs
              tail
4          | _ -> []
```

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.

I will call this function `createLineList`

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

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

This is because i couldn't figure out how to account for uneven group splitting, meaning there is 8 and 9, are not in a group.

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.

```
1      let createLineList (students : int list) (lineNum : int) =
2          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 i can deals with cases of different lengths, instead of having to hard code each and every case. You split the problem into smaller bits.