For this exercise it is important to remember that while iterative solutions need extra parameters to function properly (accumulators or continuations) these should be hidden from the end user. The easiest way to do this is using nested functions as demonstrated in class.

This has two main advantages

1. It hides unnecessary implementation details from the user
2. The inner function can take any amount of arguments in any order you wish depending on the problem at hand

You can, however, if you wish define an auxiliary function and place it outside the main function. Regardless, in places where we mention *auxiliary* function in the exercises below then we are referring to the iterative function. Also, for mutually recursive functions you will need top-level auxiliary functions.

Exercise 5.1

(based on Exercise 9.3 from HR)

In the first assignment you created the function `sum : int * int -> int` where

$$sum(m, n) = m + (m + 1) + (m + 2) + \ldots + (m + (n - 1)) + (m + n)$$

Declare an iterative solution `sum : int -> int -> int` to this problem (do note that we remove the tuple argument here as that is generally bad style).

Exercise 5.2

(based on Exercise 9.4 from HR)

Provide an iterative solution for `List.length`, `length : 'a list -> int` that given a list `lst` returns the length of `lst`.

Exercise 5.3

The library function `List.fold` enjoys having a very simple and straightforward iterative definition

```
let rec fold folder acc =
    function
    | []      -> acc
    | x :: xs -> fold folder (folder acc x) xs
```

The library function `List.foldBack`, however, is not as clear cut, as the most obvious solution is not iterative.

```
let rec foldBack folder lst acc =
    match lst with
    | []      -> acc
    | x :: xs -> folder x (foldBack folder xs acc)
```

Write an iterative function using a continuation `foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` that given a folder function `f`, a list `[x1, x2, ..., xn]` and an accumulator `acc` returns `f x1 (f x2 (f ... (f xn acc) ... ))`.

You may not reverse the list and call `fold`

Exercise 5.4

(based on Exercise 9.6 from HR)

An iterative version of the factorial function that uses accumulators looks as follows:

```
let factA x =
    let rec aux acc =
        function
        | 0 -> acc
        | x -> aux (x * acc) (x - 1)

    aux 1 x
```

Define an iterative function `factC : int -> int` that uses continuations and compare the running time between the two. Which solution is faster and why?

Exercise 5.5

(based on exercise 9.7 from HR)

A solution to the fibonacci sequence encoded with while-loops (this is a truly horrible idea and you should never use loops to compute values) looks as follows:

```
let fibW x =
    let mutable res1 = 0
    let mutable res2 = 1
    let mutable i = 1
    while (i <= x) do
        let temp = res1
        res1 <- res2
        res2 <- temp + res2
        i <- i + 1
    res1
```

Write two iterative functions `fibA : int -> int`, which uses two accumulators similarly to the while loop above to compute its answer, and `fibC : int -> int` that uses a continuation. Compare the running times of the three functions.

Hint: The auxiliary function for `fibA` should have type `int -> int -> int -> int` (the input value, the two accumulators, and the result).
Hint: The auxiliary function for `fibC` should have type `int -> (int -> int) -> int` (the input value, the continuation function, and the result).

Exercise 5.6

(based on Exercise 9.10 from HR)

Do this exercise and understand this exercise before proceeding with the next exercise

Consider the following list-generating function:

```
let rec bigListK c =
    function
    | 0 -> c []
    | n -> bigListK (fun res -> 1 :: c res) (n - 1)
```

The call `bigListK id 130000` causes a stack overflow. Analyse the problem and describe exactly why this happens. Why is this **not** an iterative function.

Exercise 5.7

In assignment 3 we worked on a domain-specific language for our tiles and our boards. In this assignment we will introduce a casting operator that lets us cast characters to their ascii-values and back.

To handle casting we change our arithmetic and character expressions to be mutually recursive.

```
type word = (char * int) list

type aExp =
    | N of int                (* Integer literal *)
    | V of string             (* Variable reference *)
    | WL                      (* Word length *)
    | PV of aExp              (* Point value lookup at word index *)
    | Add of aExp * aExp      (* Addition *)
    | Sub of aExp * aExp      (* Subtraction *)
    | Mul of aExp * aExp      (* Multiplication *)
    | CharToInt of cExp       (* NEW: Cast to integer *)

and cExp =
   | C  of char               (* Character literal *)
   | CV of aExp               (* Character lookup at word index *)
   | ToUpper of cExp          (* Convert character to upper case *)
```

```
    | ToLower of cExp         (* Convert character to lower case *)
    | IntToChar of aExp       (* NEW: Cast to character *)
```

Create mutually recursive, but not tail recursive, functions `arithEvalSimple : aExp -> word -> Map<string, int> -> int` and `charEvalSimple : cExp -> word -> Map<string, int> -> char` that given an arithmetic expression `a` or a character expression `c`, a word `w`, and a state `s`, evaluates `a` or `c` with respect to `w` and `s` respectively.

Exercise 5.8

The functions from exercise 5.7 are not tail recursive and will overflow the stack for large expressions.

Create the mutually tail-recusive functions `arithEvalTail : aExp -> word -> Map<string, int> -> (int -> 'a) -> 'a` and `charEvalTail : cExp -> word -> Map<string, int> -> (char -> 'a) -> 'a` that have the same specification as the exercise from 5.7 except that they also take a continuation which is necessary for a tail-recursive implementation. Note that the type of the continuation for `arithEvalTail`, for example, is `int -> 'a` rather than `int -> int` as in previous examples. This is necessary for the tail recursion as we must be able to cast the continuation for both integer and character evaluation (details are in the slides from the lecture).

To create the top-level evaluation functions the folowing definitions then suffice:

```
let arithEval a w s = arithEvalTail a w s id
let charEval c w s  = charEvalTail c w s id
```