TDT4165

# Assignment 3

Karoline Barstein

All code is attached in the .zip file xxxx.

## Task 1

a) When `A = 2`, `B = 1` and `C = -1`, then `RealSol = true`, `X1 = 0.5` and `X2 = -1`.
When `A = 2`, `B = 1` and `C = 2`, then `RealSol = false`, while `X1` and `X2` stays unbound.

b) **Why are procedural abstractions useful? Give at least two reasons.**
Procedural abstraction allows one to separate out pieces of code in procedures or functions, and then call the procedures/functions when we want to execute that piece of code. It is useful because it prevents redundancy, and it makes the code more readable (if the procedures are named well) and easier to debug. It also makes it possible to separate out pieces of code that are used several times, and then allow the procedure to take in different parameters (for different, but similar cases). An example of the latter is the `QuadraticEquation` procedure in part a). Calculating the roots of a quadratic equation follows the same algorithm for any numbers A, B and C. Thus, it is highly useful to wrap this piece of code in a procedure because it allows us to call the procedure every time we want to calculate the roots rather than repeat the same lines of code several times.

c) **What is the difference between a procedure and a function?**
Generally speaking; The application of a procedure is a statement, i.e., it has no return value. It executes a set of commands based on input parameters (which can be zero or many). The application of a function is an expression, i.e., it has a return value. Functions are used to calculate a value based on the value of input parameters (which can be zero or many).

## Task 3

a) See the .zip file.

b) See comments in the RightFold.oz file.

c) See the .zip file.

d) **For the `Sum` and `Length` operations, would left fold (a left-associative fold) and right fold give different results? What about subtraction?**
For associative operations, like the operations in the `Sum` and `Length` functions (which both are additive), the order of which the operations are executed does not matter. Hence, right fold and left fold will give the same result. Subtraction, however, is not an associative operation. We will get different results choosing right fold or left fold. An example follows, using the list [1 2 3 4].
Right fold will calculate

$$(0-(1-(2-(3-4)))) = (0-(1-(2-3+4))) = (0-(1-2+3-4)) = 2 \quad (1)$$

while left fold will calculate

$$((((0-1)-2)-3)-4) = (((0-1-2)-3)-4) = ((0-1-2-3)-4) = (0-1-2-3-4) = -10 \quad (2)$$

e) **What is a good value for `U` when using `RightFold` to implement the product of list elements?**
The neutral/identity element for a product should be 1. This element should not change the value of the expression when it is applied by the given operation.

# Task 5

a) See the .zip file.

b) **Give a high-level description of your solution and point out any limitations you find relevant.**
The lazy number generator has a start value as its only argument. A list is created, where the given start value takes place as the first element in the list. The second (or rest) element is an anonymous function. When called, this function will execute the lazy number generator again (implicit recursively), with a start value increased by 1. This means that once called, the number of actual created elements of the list will increase by 1, and so on. This allows us to make an "infinite" list, because the wrapping of the recursive call in a function that takes place as a part of the list makes the function only generate elements when they are explicitly asked for. The final list that is returned contains all the elements that has been explicitly called for, and the rest of the list is always represented by the anonymous function.

There are some limitations in the solution. To find (only) an element number $n$ ($n$ could be big) in the list, you will have to type a long expression; `{{[...]{{{{LazyNumberGenerator}.2}.2}.2}.2 [...]}.2}.1`. This will also trigger many calculations/calls to the function that creates the next element of the list, even though we just need the $n$-th element (but as I have understood, this also happens when using the `lazy` keyword). Compared to writing a similar function using the `lazy` keyword, we e.g. could eliminate writing all the curly braces (except the ones around the "base" function call) if using the built-in keyword, and the function body would be a bit shorter (we would eliminate the inner function, and use an explicit recursive call).

# Task 6

a) **Is your Sum function from Task 2 tail recursive? If yes, explain why. If not, implement a tail recursive version and explain how your changes made it so.**
A function is tail recursive if it calls itself at the end of the function, i.e., it does no computation after the recursive call is executed. The Sum function from task 2 may seem tail recursive at first, but actually an operation (an addition of List.1) is done after (or to the result of) the recursive call, i.e., the function needs to store more than just the returned value from the previous call. Hence, it is not tail recursive. A tail recursive version, `SumTail`, is written in the SumList.oz file. An inner function which takes an accumulator as an additional argument is "wrapped" inside the `SumTail` function. This allows us to store the current result in the accumulator, and increase the accumulator for each recursive call. This eliminates the operation on the result of each recursive call, and the function only needs to remember the result of the previous recursive call. I.e., the last call inside the function is simply just the recursive call, and hence the function is tail recursive.

b) **What is the benefit of tail recursion in Oz?**
In Oz, last call optimization is automatically applied to tail-recursive procedures. This means, we avoid allocating a new stack frame for the function. We achieve this because the calling function just returns the value it got from the recursive function call.

c) **Do all programming languages that allow recursion benefit from tail recursion? Why/why not?**
No, in e.g. Java and Python, there is no last call optimization, so there will be no extra benefit from tail recursion.